

目錄

逆向工程入门指南

Reverse Engineering for Beginners



Dennis Yurichev <dennis(a)yurichev.com>

《Reverse-Engineering-for-Beginners》中文翻

译版一些说明

- 本分支是基于乌云所翻译的《RE4B》所衍生的翻译版本。经过乌云允许后，我们fork了它并进行一些错误的修复和更新。
- 人邮社出版的《RE4B》翻译版与本分支无任何关系，是由Archer和Anti团队一起翻译而成。
- 和人邮社沟通后，因翻译版权问题，本分支只会对已翻译的内容进行错误修复。不会再对作者的主干分支进行同步更新。
- 有兴趣的可以加QQ群一起交流逆向工程：565270515
- 请勿再fork本分支进行传播

参与过的翻译人员（如有遗漏烦请提醒一声）

- 瞌睡龙、糖果、blast、magix526、Larryxi、左懒、DM_、Zing、inkydragon、xqin

Part I 代码模式

Part I 代码片段

> Everything is comprehended in comparison - Author unknown

我在开始学C/C++的时候，经常写一些小段的代码编译一下，然后观察输出的汇编代码。这种习惯让我很容易理解代码背后到底发生了什么。这种习惯让C/C++代码和编译器产生的汇编语言的关系深深地印在我的脑海里，对我来说很容易就能通过汇编代码想出C代码和函数粗略的样子。或许这个技巧对其他初学者能有所帮助。

本书有时候会用到一些旧的编译器，这是为了尽可能得到最短的(最简单)代码片段。

关于练习

作者在学习C语言的时候，经常些写一些C语言的小函数，然后逐渐的将他们重写成汇编语言，并尝试让代码尽可能的短。现在这种做法不是很值得提倡，因为很难在效率上和现代编译器相竞争。不过这是一种深入理解汇编语言的好方法。因此，你可以放轻松一些，随便在这本书里找一段汇编代码，然后尝试者让它更短一些。当然不要忘记测试你所写的汇编代码。

优化等级和调试信息

源代码可以用不同的编译器,以不同的优化等级来编译。 典型的编译器有三种优化等级，其中0级代表不优化。优化既可以针对代码的体积，也可以针对代码的运行速度。一个无优化的编译器编译会更快一点，生成的代码也更容易理解一些(虽然很冗长)。反之一个带优化的编译器会运行的更慢，并编译出运行的更快的代码(但代码并不会更紧凑)。

除了优化的级别和方向外，一个编译器还可能在生成的文件里包含一些调试信息，这样的代码更容易调试。个编译器的一个重要特性是，在输出文件里面，可能会有源代码到机器码地址的连接。另一方面，带优化的编译器，更倾向于将所有的源代码优化掉后再输出，因此源代码不会出现在输出的机器码里。一个逆向工程师这两种情况都有可能遇到，因为有的开发者会打开优化，有的不会。所以，在这本书里,我们会尽可能关注每个例子的调试和发行版本的代码特征。

第一章

CPU简介

CPU是一种可以执行由机器码组成的程序的设备。

词汇表：

Instruction：用于控制CPU的指令。最简单的例子有：在寄存器之间进行数据转移操作，内存操作，算术操作。原则上每种CPU会有自己独特的一套指令构架(Instruction Set Architecture(ISA))。

Machine code: 机器码，CPU能直接处理的代码。每条指令都会被译成几个字节的机器码。

Assembly Language: 汇编语言，助记码和其他一些像宏那样的扩展组成的、便于程序员编写的语言。

CPU register：CPU寄存器，每个CPU都有一些通用寄存器(General Purpose Registers(GPR))。X86有8个，x86-64(amd64)有16个，ARM有16个，最简单的理解寄存器的方法就是，把寄存器想成一个未指定类型的临时变量。想象你在使用高级语言编程，并且只能用8个32bit(或 64-bit)的变量。但是只用这些可以完成非常多的事情。

那么你可能想知道，机器码跟程序语言有什么区别呢？答案主要在于人类和CPU的思维方式并不类似。对于人类来讲，使用例如C/C++, Java, Python这样高级语言会比较简单，但是CPU更喜欢低级抽象的东西。也许有一天CPU也能直接执行高级语言的语句，但那样的CPU肯定会变得比天的要复杂好几倍。类似的，人类之所以使用汇编语言会感觉不很方便，是因为它非常的低级，而且很难用它写很长的代码而不出错。

将高级语言转换成汇编语言的程序，被称为编译器。

1.1 关于不同指令集的几点

x86构架一直都带有可变长度的操作码，因此64位的世纪到来时，x64的扩展对这个构架并没有太大的影响。事实上x86构架还包含着很多最早在16位8086 CPU里出现的指令，他们在今天的处理器中依旧可以被使用。

ARM是一种带定长操作符的精简指令集的CPU，它过去有很多优点。

最初，ARM所有的指令都被编码为4个字节。这现在被称为“ARM模式”。但是人们发现这样做并不像他们一开始所想的那样节约。事实上现实中最常用的CPU指令都可以用更少的字节来编码。因此他们又增加了一种每个指令只以2字节编码的构架，叫做Thumb。这现在被称为“Thumb模式”。然而并不是所有的ARM构架都能被编码为2字节，所以Thumb构架在某些方面是有限制的。值得注意的是以ARM模式或Thumb模式编译得代码，有可能同时出现在一个程序里。

ARM的设计记者认为Thumb可以作为一种扩展存在，这就产生了Thumb-2，它首次在ARMv7里出现。Thumb-2依旧使用2字节的指令集，但是它也有一些4字节的新指令。有一种很常见的错误观念，认为Thumb-2是ARM和Thumb的混合物。这是不对的，Thumb-2扩展了对所有处理器特性的支持，所以他可以和ARM模式相竞争———很显然这个目标被很好的实现了。主要的iPod/iPhone/iPad应用是用Thumb-2指令集编译的(公认的，这主要是因为Xcode把这个设为默认模式)。

之后，64位的ARM发布了，这种构架有4字节的操作码，而且也不需要任何附加的Thumb模式。即便如此64位的要求也影响了构架，导致了现在有3种ARM构架：ARM 模式、Thumb 模式(包括Thumb-2)和ARM64。这些构架有部分交叉，可以说他们是不同的构架，但不能说他们是一个构架的的不同变种。因此在这本书里，我们会试着加入全部三种ARM构架的代码片段。

顺带提一下，还有很多带32位变长操作码的、精简指令集的构架。例如：MIPS,PowerPC and Alpha AXP.

第二章

最简单的函数

最简单的函数可能只需要返回一个常数值。 这有个例子： 清单 2.1: C/C++ 代码

```
int f()
{
    return 123;
};
```

让我们编译一下！

2.1 x86

下面是带优化的GCC和MSVC编译器在x86平台上的输出： 清单 2.2: 带优化的 GCC/MSVC (汇编输出)

```
f:
    mov     eax, 123
    ret
```

这里只有两个函数：第一个把123放入 `EAX` 寄存器里， `EAX` 通常被用作存放函数的返回值。第二个是 `RET` , `RET` 把控制权交给主调函数。

主调函数会从 `EAX` 里取出返回值。

2.2 ARM

在ARM平台上会有一点点区别。

清单 2.3: 带优化的 Keil 6/2013 (ARM mode) ASM 输出

```
f PROC
        MOV     r0,#0x7b ; 123
        BX         lr
        ENDP
```

ARM 用 `R0` 来储存函数的返回值，所以123被复制进 `R0` .

在ARM构架里返回值的地址不是保存在局部堆栈里，而是放在链接寄存器里，所以 `BX LR` 指令转跳到那个地址，这有效地把控制权转交给了主调函数。

值得注意的是，对于x86和ARM构架来说， `MOV` 是一个容易令人误解性的名称。数据事实上没有被移动，而是被复制了。

MIPS

在MIPS的世界里有两种寄存器命名的形式：用数字(从 `$0` 到 `$31`)或者用别名(`$V0` , `$A0` , 等等)。

GCC汇编输出中会像下面列表中那样用数字表示寄存器：

清单 2.4: 带优化的 GCC 4.4.5 (汇编输出)

```
j    $31
li    $2,123 # 0x7b
```

而IDA会把它转换成别名： 清单 2.5: 带优化的 GCC 4.4.5 (IDA)

```
jr      $ra
li      $v0, 0x7B
```

`$2` (或 `$V0`)被用来储存函数的返回值。 `LI` 代表“立即加载”，这也是MIPS里 `MOV` 的一个的等价用法。

剩下的指令是转跳指令(`J` 或 `JR`)，它把控制权交给主调函数，并转跳到 `$2` (或 `$V0`)寄存器里的地址。

这个寄存器类似于 `ARM` 里的LR寄存器。

你可能想知道为什么加载指令(`LI`)和转跳指令(`J` 或 `JR`)的位置被交换了。这都是由于RISC中被称为“分支延迟槽”的特性。

对于这种现象发生的原因有个借口：这是一些MIPS构架编译器的一个怪癖。但这对我们的目的来说并不重要，我们只需记住在MIPS里是这样的：在转跳指令之后的指令会先比转跳指令本身先执行。

2.3.1 关于MIPS指令/寄存器命名的一点

在MIPS的世界里，寄存器和指令名习惯上使用小写。但为了一致性，我们坚持使用大写，并在这本书里，把这点当做一个其他编译器都遵守的约定。

第三章

Hello,world!

让我们用《C语言程序设计》中最著名的例子开始吧[Ker88]：

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

3.1 x86

3.1.1 MSVC

让我们在MSVC 2010中编译一下：

```
cl 1.cpp /Fa1.asm
```

（/Fa 选项表示让编译器生产汇编代码文件）

代码清单 3.1: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push     ebp
        mov     ebp, esp
        push     OFFSET $SG3830
        call     _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

MSVC生成的汇编代码用的是Intel的汇编语法。Intel语法与AT&T语法的区别将会在3.1.3讨论。

编译器会生成连接到 `1.exe` 的 `1.obj` 文件。在我们的例子当中，该文件包含两个部分： `CONST` （放数据常量）和 `_TEXT` （放代码）。

字符串 `"hello, world"` 在C/C++的类型为 `const char[]` [Str13, 7.3.2],，然而它没有自己的变量名。编译器需要处理这个字符串，于是就自己给他定义了一个内部名称 `$SG3830` 。

所以我们的例子可以重写为下面这样：

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

让我们回到汇编代码，正如我们看到的，字符串是由0字节结束的，这是标准的C/C++字符串。关于C/C++字符串见：57.1.1

在代码部分， `_TEXT` ，那儿只有一个函数：main()。main()函数与大多数函数一样都由起始代码开始，由收尾代码结束。

函数当中的起始代码结束以后，我们看见了对printf()函数的调用： `CALL _printf` 。在调用之前，保存我们问候语字符串的地址（或指向它的指针），已经在PUSH指令的帮助下，被存放在栈中。

当printf()函数执行完返回到main()函数的时候，字符串地址(或指向它的指针)仍然在堆栈中。当我们完全不需要它的时候，堆栈指针（ESP寄存器）需要被复原。

`ADD ESP, 4` 意思是ESP寄存器的值加4。

为什么是4呢？因为这是32位的程序，通过栈传送地址刚好需要4个字节。如果是64位的代码则需要8字节。 `ADD ESP, 4` 在效率上等同于 `POP register` ，但是后者不需要使用任何寄存器。

一些编辑器（如Intel C++编译器）在同样的情况下可能会用 `POP ECX` 代替ADD（这样的模式可以在Oracle RDBMS代码中看到，因为它是由Intel C++编译器编译的），这两条指令的效果基本相同，但是ECX的寄存器内容会被改写。Intel C++编译器可能用 `POP ECX` ，因为这条指令比 `ADD ESP, X` 更短，（ `POP` ——1字节对应 `ADD` ——3字节）。

这里有一个在Oracle RDBMS中用 `POP` 而不用 `ADD` 的例子。 清单 3.2: Oracle RDBMS 10.2 Linux (app.o 文件)

```
.text:0800029A          push     ebx
.text:0800029B          call     qksfroChild
.text:080002A0          pop     ecx
```

在调用printf()之后，原来的C/C++代码执行 `return 0` ，返回0当做main()函数的返回结果。在生成的代码中，这被编译成指令 `XOR EAX, EAX` 。

XOR事实上就是异或，但是编译器经常用它来代替 `MOV EAX, 0` 原因就是它需要的字节更短（ `XOR` 需要2字节对应 `MOV` 需要5字节）。

有些编译器则用 `SUB EAX, EAX` 就是EXA的值减去EAX，也就是返回0。

最后的 `RET` 指令返回控制权给调用者。通常这是C/C++的库函数代码，它会按顺序，把控制权返还给操作系统。

3.1.2 GCC

现在我们尝试在linux中用GCC 4.4.1编译同样的C/C++代码

`gcc 1.c -o 1`

下一步，在IDA反汇编的帮助下，我们看看main()函数是如何被创建的。IDA与MSVC一样，也是使用Intel语法。

代码清单 3.3:IDA里的代码

```
main proc near
var_10          = dword ptr -10h
    push ebp
    mov  ebp, esp
    and  esp, 0FFFFFFF0h
    sub  esp, 10h
    mov  eax, offset aHelloWorld ;` `"hello, world"
    mov [esp+10h+var_10], eax
    call _printf
    mov eax, 0
    leave
    retn
main            endp
```

结果几乎是相同的， `"hello,world"` 字符串地址（保存在data段的）一开始保存在EAX寄存器当中，然后保存到栈当中。

此外在函数开始我们看到了 `AND ESP, 0FFFFFFF0h` ，这条指令将ESP寄存器中的值对齐为16字节。这让堆栈中的所有值，都以相同的方式对齐。(如果分配的内存地址大小被对齐为4或16字节，CPU的性能会更好。)

`SUB ESP，10H` 在栈上分配16个字节。 虽然在下面我们可以看到，这里其实只需要4个字节。

这是因为分配的堆栈的大小也被对齐为16位。

该字符串的地址（或这个字符串指针）直接存入到堆栈空间，而不使用PUSH指令。 `var_10` ，是一个局部变量，也是 `printf()` 的参数。

然后 `printf()` 函数被调用。

不像MSVC，当gcc编译不开启优化时，它使用 `MOV EAX，0` 清空EAX，而不用更短的指令。

最后一条指令， `LEAVE` 相当于 `MOV ESP，EBP` 和 `POP EBP` 两条指令。换句话说，这相当于将堆栈指针（ESP）恢复，并将EBP寄存器复原到其初始状态。

这是很有必要的，因为我们在函数的开头修改了这些寄存器的值（ESP和EBP）（执行 `MOV EBP，ESP` / `AND ESP, ...` ）。

3.1.3 GCC:AT&T 语法

我们来看一看在AT&T当中的汇编语法，这个语法在UNIX类系统当中更普遍。 代码清单 3.4: 让我们用 GCC 4.7.3 编译 `gcc -S 1_1.c`

我们将得到这个：

代码清单 3.5: GCC 4.7.3

```
        .file   "1_1.c"
        .section    .rodata
.LC0:
        .string "hello, world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl     %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl     %esp, %ebp
        .cfi_def_cfa_register 5
        andl     $-16, %esp
        subl     $16, %esp
        movl     $.LC0, (%esp)
        call     printf
        movl     $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack, "", @progbits
```

这段代码包含了很多的宏（以点开始）。目前我们不关心这个。

现在为了简单起见，我们先不看这些。（除了 .string ，就像C-string一样，用于编码一个以null结尾的字符序列）。然后，我们将看到这个：

代码清单 3.6: GCC 4.7.3

```
.LC0:
    .string "hello, world"
main:
    pushl   %ebp
    movl %esp, %ebp
    andl $-16, %esp
    subl $16, %esp
    movl $.LC0, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
```

在Intel与AT&T语法当中一些主要的区别就是：

- 操作数写在后面
  在Intel语法中：\\\
  在AT&T语法中：\\\
  有一个简单的记住它们的方法: 当你面对intel语法的时候，你可以想象把等号(=)放到2个操作数中间，当面对AT&T语法的时候，你可以放一个右箭头(→）到两个操作数之间。
- AT&T: 在寄存器名之前需要写一个百分号(%)并且在数字前面需要加上美元符($)。并用圆括号替代方括号。
- AT&T: 以下是一些添加到操作符后，用来表示数据形式的后缀：
  – q — quad (64 bits)
  – l — long (32 bits)
  – w — word (16 bits)
  – b — byte (8 bits)

让我们回到上面的编译结果：它和在IDA里看到的是一样的。只有一点不同： `0FFFFFFF0h` 被写成了 `$-16` 。但这是其实是一样的，10进制的16在16进制里表示为0x10。-0x10就等同于0xFFFFFFF0(针对于32位的数据类型)。

另外：返回值通常用 `MOV` 置0，而不用 `XOR` 。MOV仅仅加载（load）了一个值到寄存器。这条指令的名称是个误称(数据没有被移动，而是被复制了)。在其他的构架上，这条指令会被称作“LOAD” 、 “STORE”或其他类似的名称。

3.2 x86-64

3.2.1 MSVC-x86-64

让我们来试试64-bit的MSVC： 代码清单 3.7: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 00H
main    PROC
    sub    rsp, 40
    lea    rcx, OFFSET FLAT:$SG2923
    call   printf
    xor    eax, eax
    add    rsp, 40
    ret    0
main ENDP
```

在x86-64里，所有的寄存器都被扩展到64位，并且名字前都带有R-前缀。这是为了减少栈的使用(即减少对外部内存/缓存的访问)，通常的做法是：用寄存器来传递参数（类似于fastcall）。也就是，一部分参数通过寄存器传递，其余的通过栈传递。

在win64里， `RCX,RDX,R8,R9` 寄存器被用来传递函数的4个参数，在这里我们可以看到指向给printf()函数的字符串的指针，没有用通过栈，而是用了 `RCX` 来传递。这些指针现在是64位的，所以他们通过64位寄存器来传递(带有R-前缀)，并且为了

向后兼容，依旧可以使用E-前缀，来访问32位的部分。

如下图所示，这是 `RAX/EAX/AX/AL` 在x86-64构架里的情况

<table>
<tr><td>7th (byte number)</td><td>6th</td><td>5th</td><td>4th</td><td>3rd</td><td>2nd</td><td>1st</td><td>0th</td></tr>
<tr><td colspan="8">RAX^{x64}</td></tr>
<tr><td colspan="4"></td><td colspan="4">EAX</td></tr>
<tr><td colspan="6"></td><td colspan="2">AX</td></tr>
<tr><td colspan="6"></td><td>AH</td><td>AL</td></tr>
</table>

main()函数会返回一个int类型的值，在C/C++里为了兼容和移植性，依旧是32位的。这就是问什么，是 `EAX` 而不是 `RAX` （即寄存器的低32位部分）在函数最后会被清0。

在寄存器里也有40字节被分配给了局部堆栈。这被称为“影子空间”。这一点我们之后会提及：8.2.1 。

3.2.2 GCC-x86-64

这次在64位的Linux里试试GCC：

代码清单 3.8: GCC 4.4.6 x64

```
    .string "hello, world"
main:
    sub     rsp, 8
    mov     edi, OFFSET FLAT:.LC0 ; "hello, world"
    xor     eax, eax  ; number of vector registers passed
    call    printf
    xor     eax, eax
    add     rsp, 8
    ret
```

在Linux,*BSD和Mac OS X里也使用同一种方式来传递函数参数。

前6个参数使用 `RDI,RSI,RDX,RCX,R8,R9` 来传递的，剩下的用栈。

所以在这个程序里，字符串指针被放到 `EDI` （RDI的低32位部分）。但是为什么不用RDI的64位部分呢？

记住这一点很重要： `MOV` 指令在64位模式下，对低32位进行写入操作的时候，会清空高32位的内容[Int13]。比如 `MOV EAX，011223344h` 将会把值写到RAX里，并且清空RAX的高32位区域。

如果我们打开编译好的对象文件(object file[.o]),我们会看到所有的指令的操作符：

代码清单 3.9: GCC 4.4.6 x64

```
.text:00000000004004D0                              main proc near
.text:00000000004004D0 48 83 EC 08           sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00        mov     edi, offset
format ;"hello, world"
.text:00000000004004D9 31 C0                 xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF        call     _printf
.text:00000000004004E0 31 C0                 xor     eax, eax
.text:00000000004004E2 48 83 C4 08           add     rsp, 8
.text:00000000004004E6 C3                    retn
.text:00000000004004E6                              main endp
```

就像看到的那样，在 `0x4004D4` 那行写入 `EDI` 花了5个字节。如果把这句换成给 `EDI` 写入64位的值，会花掉7个字节。显然，GCC在试图节省空间，除此之外，数据段(data segment)中包含的字串不会分配到高于4GiB的地址。

可以看到在调用printf()函数前， `EAX` 被清空了，这是因为在x86-64的 `*NIX` 系统上， 使用过的向量寄存器的数量会被存入 `EAX` [Mit13]。

3.3 关于GCC 额外的一点

(3.1.1)，并且匿名的C字符串带有常量的类型C字符串在常量段被分配的地址是一定不变的。基于这样的事实，就有一个有趣的结论：编译器可能只用了字符串的某一部分。

让我们看看这个例子：

```
#include <stdio.h>

int f1()
{
    printf ("world\n");
}
int f2()
{
    printf ("hello world\n");
}
int main()
{
    f1();
    f2();
}
```

一般的C/C++编译器(包括MSVC)会分别分配给地址两个字符串，但是让我们看看GCC干了什么： 代码清单 3.10: GCC 4.8.1 + IDA listing

```
f1          proc near

s          = dword ptr -1Ch
        sub     esp, 1Ch
        mov     [esp+1Ch+s], offset s ; "world\n"
        call     _puts
        add     esp, 1Ch
        retn
f1          endp

f2          proc near

s          = dword ptr -1Ch

        sub     esp, 1Ch
        mov     [esp+1Ch+s], offset aHello ; "hello "
        call    _puts
        add     esp, 1Ch
        retn
f2          endp

aHello      db 'hello '
s          db 'world',0xa,0
```

实际上，当我们打印"hello world"字符串时，这两个单词被放在内存里相邻的位置。函数f2()中调用的 `puts()` 并不知道字符串已经被分开了。事实上字符串并没有被真正分开，只是在代码里被假装分开了。

当 `puts()` 被f1()调用时，他使用“world”字符串加上一个0，`puts()` 并不清楚字符串之后还有什么。

这个聪明的小技巧至少在GCC里被经常使用，他能够节省一些内存。

3.4 ARM

作者根据自身对ARM处理器的经验，选择了几款流行的编译器：

- 嵌入式领域很流行的：Keil Release 6/2013
- 苹果的Xcode 4.6.3 IDE(其中使用了LLVM-GCC4.2编译器)
- GCC 4.9 (Linaro) (for ARM64) 可用的32位.exe

32位ARM的代码(包括Thumb 和 Thumb-2模式)被用在这本书的所有例子里，如果未做其他提示，我们谈论64位ARM时会叫它ARM64.

3.3.1 未进行代码优化的Keil 6/2013 编译：ARM模式

让我们在Keil里编译我们的例子

```
armcc.exe –arm –c90 –O0 1.c
```

armcc编译器可以生成intel语法的汇编程序列表，但是里面有高级的ARM处理器相关的宏，对我们来讲更希望看到“指令原来的样子”，所以让我们看看IDA反汇编之后的结果。

代码清单 3.11: 无优化的 Keil 6/2013 (ARM 模式) IDA

```
.text:00000000                  main
.text:00000000 10 40 2D E9          STMFD     SP!, {R4,LR}
.text:00000004 1E 0E 8F E2         ADR     R0, aHelloWorld ; "h
ello, world"
.text:00000008 15 19 00 EB         BL         __2printf
.text:0000000C 00 00 A0 E3         MOV     R0, #0
.text:00000010 10 80 BD E8         LDMFD     SP!, {R4,PC}

.text:000001EC 68 65 6C 6C +aHelloWorld  DCB "hello, world",0 ;
DATA XREF: main+4
```

在例子中，我们可以发现所有指令都是4字节的，因为我们编译的时候选择了ARM模式，而不是Thumb模式。

最开始的指令是 `STMFD SP!, {R4, LR}` ，这条指令类似x86平台的 `PUSH` 指令，它会把2个寄存器（R4和LR）的值写到栈里。不过为了简化，在 `armcc` 编译器输出的汇编代码里会写成 `PUSH {R4, LR}` ，但这并不准确，因为 `PUSH` 命令只在Thumb模式下可用，所以为了减少混乱，我们用IDA来做反汇编工具。

这指令首先会减少 `SP` 的值，这样它在栈中指向的空间就被释放，以留给新条目使用，然后将R4和LR的值存入被修改后的 `SP` 的储存区域中。

这条指令（类似于Thumb模式的PUSH）允许一次压入好几个寄存器的值，非常实用。顺带说一下，在x86里面它没有等价的指令。还有一点值得注意的是： `STMFD` 指令是广义的 `PUSH` 指令(扩展了它的功能)，因为他能操作任何寄存器，不只是 `SP` 。换句话说， `STMFD` 可以用于将一组寄存器储存在特定的内存地址上。

`ADR R0, aHelloWorld` 这条指令将 `"hello, world"` 字串的地址偏移加上或减去PC寄存器的值。有人会问， `PC` 寄存器在这里有什么用呢？这被称作浮动地址码（position-independet code），这样的代码可以在内存中非固定的地址上运行。换句话说，这是和 `PC` 寄存器相关的寻址。ADR这条指令，考虑了指令的地址和字符串真正所在的地址的差异。无论操作系统把我们的代码加载到哪里，这个差值(偏移)总是相同的。这也是为什么，我们每次都要加上当前的指令地址(从 `PC` 里)，以获取内存中字串的绝对地址。

`BL __2print` 这条指令用于调用printf()函数，以下是这条指令是如何工作的：

- 将BL指令（0xC）后面的地址写入LR寄存器；
- 然后把printf()函数的入口地址写入PC寄存器，将控制权交给printf()函数。

当printf()函数执行完之后，它必须知道该把控制权返回谁。这就是为什么，每个函数都会把控制权交给 `LR` 寄存器中的地址。

函数返回地址的存放位置，也正是“纯”-RISC处理器（例如ARM）和CISC处理器(例如x86)的区别。

另外，一个32位地址或者偏移量不能被编码到32位BL指令里，因为BL指令只有24位的空间。我们应该还记得，所有的ARM模式下的指令都是4字节的（32位）。因此，指令占用了4位的地址。这也就意味着最后2bits(这里总会被设置成0)被忽略了，总的来说，我们有26位可用于偏移编码。这足够去访问大约 `当前_PC` ±32M的地址。

下面我们来看 `MOV R0， #0` 这条语句，这条语句就是把0写入R0寄存器。这是因为C函数返回了0，返回值会放在R0里。

最后一条指令是 `LDMFD SP!, R4,PC` ，这是STMFD的逆指令。为了将初始值存入 `R4` 和 `PC` 寄存器里，这条指令会从栈上(或任何其他的内存区域)读取保存的值，并且增加堆栈指针 `SP` 的值。这非常类似x86平台里的 `POP` 指令。

最前面那条 `STMFD` 指令，将 `R4` ，和 `LR` 寄存器成对保存到栈中。在 `LDMFD` 执行的时候， `R4` 和 `PC` 会被复原。

我们已经知道，函数的返回地址会保存到 `LD` 寄存器里。第一条指令会把他先保存到栈里，这是因为main()调用printf()函数时，会使用LD寄存器。在函数的最后，这个值会被直接写入 `PC` 寄存器，完成函数的返回操作。

因为在C/C++里 `main()` 一般是主函数，控制权会返回给系统加载器或者CRT里面的指针或其他类似的东西。

所有的这些都允许在函数的结尾忽略 `BX LR` 指令。

汇编代码里的 `DCB` 关键字用来定义ASCII字串数组，就像x86汇编里的 `DB` 关键字。

3.4.2未进行代码优化的Keil 6/2013 编译： (Thumb模式)

让我们用下面的指令，将相同的例子用Keil的Thumb模式来编译一下。

```
armcc.exe –thumb –c90 –O0 1.c
```

我们可以在IDA里得到下面这样的代码： 代码清单 3.12: Non-optimizing Keil 6/2013 (Thumb mode) + IDA

```
.text:00000000             main
.text:00000000 10 B5          PUSH        {R4,LR}
.text:00000002 C0 A0          ADR        R0, aHelloWorld ; "hell
o, world"
.text:00000004 06 F0 2E F9    BL         __2printf
.text:00000008 00 20          MOVS        R0, #0
.text:0000000A 10 BD          POP         {R4,PC}
.text:00000304 68 65 6C 6C +aHelloWorld  DCB "hello, world",0 ;
DATA XREF: main+2
```

我们首先就能注意到指令都是2字节(16位)的了，这正是Thumb模式的特征。

但BL指令是2由个16位的指令来构成的。因为不可能只用16位操作符里的小空间，去加载printf()的偏移量。因此，第一个16位指令，用来加载函数偏移的高10位，第二个指令加载函数偏移的低11位。正如我说过的，所有的Thumb模式下的指令都是2字节(16位)的。这就意味着一个Thumb指令，无论如何不可能在奇数位的地址上。基于以上因素，地址的最后一位将会在编码指令时省略。总的来讲， `BL` 在Thumb模式下可以访问 `当前_PC` ±2M的地址。

至于在这个函数中的其他指令: `PUSH` 和 `POP` ，它们跟上面讲到的 `STMFD/LDMFD` 很类似，但这里不需要指定 `SP` 寄存器， `ADR` 指令也跟上面的工作方式相同。 `MOVS` 指令将函数的返回值0写到了 `R0` 寄存器里，让函数返回0。

3.4.3 开启代码优化的Xcode（LLVM）(ARM模式)

Xcode 4.6.3不开启代码优化的情况下，会产生非常多冗余的代码，所以我们学习一个优化过的版本。这个版本所用的指令的数量会尽可能的少。

开启 `-O3` 编译选项

Listing 3.13: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
__text:000028C4         _hello_world
__text:000028C4 80 40 2D E9        STMFD   SP!, {R7,LR}
__text:000028C8 86 06 01 E3        MOV     R0, #0x1686
__text:000028CC 0D 70 A0 E1     MOV     R7, SP
__text:000028D0 00 00 40 E3     MOVT    R0, #0
__text:000028D4 00 00 8F E0     ADD     R0, PC, R0
__text:000028D8 C3 05 00 EB     BL      _puts
__text:000028DC 00 00 A0 E3     MOV     R0, #0
__text:000028E0 80 80 BD E8     LDMFD   SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C +aHelloWorld_0    DCB "Hello worl
d!", 0
```

我们已经非常熟悉 `STMFD` 和 `LDMFD` 指令了，这里就跳过不讲。

下一条， `MOV` 指令就是将数字 `0x1686` 写入 `R0` 寄存器里。这个值是字符串"Hello world！"的指针偏移量。

`R7` 寄存器(在[App10]里这是个标准)是一个帧指针，在之后的章节我们会介绍它。

`MOVT R0， #0` (`MOVe Top`)指令时向寄存器 `R0` 的高16位写入0。这是因为在ARM模式下， `MOV` 这条指令，只对低16位进行操作。记住！在ARM模式下，所有的指令都被限定在32位以内。当然这个限制并不影响，数据在2个寄存器之间的直接的转移。这也是 `MOVT` 这种向高16位(包含第16~31位)写入的附加指令存在的意义。但在这里它其实是多余的，因为 `MOVS R0，#0x1686` 这条指令也能把寄存器的高16位清0。这或许就是相对于人脑来说编译器的不足。

`ADD R0，PC，R0` 指令把 `PC` 寄存器的值相到 `R0` 里，用来计算 `"Hello world!"` 字符串的绝对地址。这如我们所知的，这里采用浮动地址码，所以这个修正还是有必要的。

`BL` 指令调用了 `puts()` 函数，而不是 `printf()` 。

GCC将第一个 `printf()` 函数替换成了 `puts()` 。因为 `printf()` 函数只有单一参数时，跟 `puts()` 函数是类似的。在大多数情况下， `printf()` 的字符串参数里，没有以 `%` 开头的特殊控制符的时候，两个函数的会输出相同的结果。如果不是这样，这两个函数的功能会有所差别。

为什么编译器会替换 `printf()` 为 `puts()` 呢？这或许是因为 `puts()` 更快一些。因为 `puts()` 只是做了字串的标准输出(`stdout`)，而不需要将字符串逐位与 `%` 相比较。

下一条语句，我们可以看到了熟悉的 `"MOV R0, #0"` 指令，用来将 `R0` 寄存器设为0。

3.4.4 开启代码优化的Xcode(LLVM)编译Thumb-2模式

在默认情况下，Xcode4.6.3会生成如下的Thumb-2代码

代码清单 3.14: 带优化的 Xcode 4.6.3 (LLVM) (Thumb-2 模式)

```
__text:00002B6C                    _hello_world
__text:00002B6C 80 B5              PUSH     {R7,LR}
__text:00002B6E 41 F2 D8 30        MOVW     R0, #0x13D8
__text:00002B72 6F 46              MOV      R7, SP
__text:00002B74 C0 F2 00 00        MOVT.W   R0, #0
__text:00002B78 78 44              ADD      R0, PC
__text:00002B7A 01 F0 38 EA        BLX      _puts
__text:00002B7E 00 20              MOVS     R0, #0
__text:00002B80 80 BD              POP      {R7,PC}

...

__cstring:00003E70 48 65 6C 6C 6F 20 +aHelloWorld DCB "Hello wor
ld!",0xA,0
```

正如我们刚刚回忆过的， `BL` 和 `BLX` 指令在Thumb模式下，被编码为一对16位的指令。在Thumb-2模式下这操作符些会被这样扩展，所以新的指令会被编码成32位的指令。很容易就能发现，Thumb-2的操作码总是以 `0xFx` 或 `0xEx` 开头。但是在IDA的反汇编代码里，操作符的位置被交换过了。对于ARM处理器来说，这是因为指令以以下方式编码：最后一个字节在最前面，接下来是第一个字符(在Thumb和 Thumb-2模式里)，对于四个字节的操作符则是：首先是第四个字节，然后是第三个，接下来是第二个，最后才是第一个字节(这是由于不同的字节序)。

下面是在IDA里，字节是如何排列的：

- ARM 和 ARM64 模式: 4-3-2-1;
- Thumb 模式: 2-1;
- Thumb-2 模式里的一对16位指令 : 2-1-4-3.

所以我们能看出来， `MOVW` ， `MOVT.W` 和 `BLX` 这几个指令都是以 `0xFx` 开始。

在Thumb-2指令里有一条是 `MOVW R0, #0x13D8` ,它的作用是将16位的值，写到 `R0` 的低16位里面，并将高位清零。

`MOVT.W R0, #0` 的作用类似与前面讲到的 `MOVT` 指令，但它工作在Thumb-2模式下。

还有些其他的差异，比如 `BLX` 指令替代了上面用到的 `BL` 指令。这样做的区别在于：这条指令除了将 `RA` 存入到寄存器 `LR` 里，还将控制全交给 `puts()` 函数，并且处理器也从Thumb/Thumb-2模式转换到了ARM模式（或者相反）。这条指令放在这里，是因为跳转到了像下面这样的位置（下面的代码以ARM模式编码）。

```
__symbolstub1:00003FEC _puts              ; CODE XREF: _hello_wo
rld+E
__symbolstub1:00003FEC 44 F0 9F E5      LDR PC, =__imp__puts
```

这本质上是个到 `puts()` 导入地址的转跳。

可能会有细心的读者要问了:为什么不在需要的时候，直接调用 `puts()` 函数呢？

因为那样做会浪费内存空间。

大多数程序都会使用额外的动态库(dynamic libraries)(Windows里面的DLL，还有*NIX里面的.so，MAC OS X里面的.dylib),经常使用的库函数会被放入动态库中，当然也包括标准C函数 `puts()` 。

在可执行的二进制文件里(Windows的PE里的.exe文件，ELF和Mach-O文件)都会有输入表段。它是一个用来引入额外模块里模块名称和符号（函数或者全局变量）的列表。

系统加载器（OS loader）会加载所有需要的模块，当在主模块里枚举输入符号的时候，会确定每个符号真正地址。

在我们的这个例子里， `__imp__puts` 就是一个系统加载器储存附加模块真正地址的32位的变量。 `LDR` 指令把这个值从变量里读取出来，并写入到 `PC` 寄存器里，并将控制权交给那个地址。

所以为了减少系统加载器完成这个过程所需的时间，最好将所有符号的地址一次性写到一个特定的地方。

另外，我们前面也指出过，我们没办法只用一条指令，并且在不访问内存的情况下，就将一个32位的值保存到寄存器里。因此，最好的办法就是，单独分出一个函数，用来在ARM模式下将控制权交给动态链接库，这样做一些类似与上面这样单一指令的函数（称做Thunk function），然后从Thumb模式里也能去调用。

在先前的例子中（以ARM模式编译的例子）， `BL` 指令也是跳转到了同一个Thunk function里。尽管没有进行模式的转变（所以指令里不存在那个”X”）。

关于形实转换函数

形实转换函数很难理解，表面上看是因为它的具有误导性的名字。

理解它最简单的方法是把他看做一个适配器，或者将一种插口转换为另一种的转换器。举个例子，一个适配器允许一个英式的电源插头插入一个美式的插座，反之亦然。

形实转换函数有时被称作封装器。

以下是对该函数的一些描述：

> P. Z. Ingerman说这个函数是"提供地址的一段代码"，他于1961年，发明了形实转换函数，并作为Algol-60 程序调用里，将实参转换为标准定义的一种方式。如果调用一个带有表达式形参的程序，编译器会生成一个形实转换函数来计算表达式的值，并将结果的地址放在某些标准位置上。 ... Microsoft 和 IBM 都在他们的基于Intel的系统里面定义了一个“16-位的环境”(带有讨厌的段寄存器和64K的内存限制)和一个“32-位的环境”(带有平坦寻址和半实时的内存管理)。 这两种环境都能在相同的电脑和操作系统上运行(感谢我们在Microsoft世界里称之为WOW的东西，WOW代表着Windows On Windows)。 MS 和 IBM都决定将16位到32位和相反的转换过程称为一个"thunk"；对于Windows 95来说，甚至有个叫做“Thunk编译器”的工具——THUNK.EXE。

(The Jargon File)

3.4.5 ARM64

GCC

让我们在ARM64 上用GCC 4.8.1编译一下这个程序。

代码清单 3.15:无优化的 GCC 4.8.1 + objdump

```
1    0000000000400590 <main>:
2     400590:     a9bf7bfd     stp     x29, x30, [sp,#-16]!
3     400594:     910003fd     mov     x29, sp
4     400598:     90000000     adrp     x0, 400000 <_init-0x3b8>
5     40059c:     91192000     add     x0, x0, #0x648
6     4005a0:     97ffffa0     bl         400420 <puts@plt>
7     4005a4:     52800000     mov     w0, #0x0
    // #0
8     4005a8:     a8c17bfd     ldp     x29, x30, [sp],#16
9     4005ac:     d65f03c0     ret
10
11      ...
12
13      Contents of section .rodata:
14      400640 01000200 00000000 48656c6c 6f210a00 ........Hello!
..
```



代码清单 3.16: main() 返回uint64_t类型的值

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
    printf ("Hello!\n");
    return 0;
}
```

结果是相似的，下面是在那一行， `MOV` 看起来是怎么样的：

代码清单 3.17: 无优化的 GCC 4.8.1 + objdump

```
4005a4:     d2800000     mov     x0, #0x0          // #0
```

3.5 MIPS

3.5.1 关于全局指针

`LDA` 负载对然后恢复了 `X29` 和 `X30` 寄存器。

3.5.2 带优化的GCC

让我们看看下面这个例子，他说明了全局指针的概念：

代码清单 3.18: 带优化的 GCC 4.4.5 (汇编输出)

```
1     $LC0:
2     ; \000 is zero byte in octal base:
3             .ascii "Hello, world!\012\000"
4     main:
5     ; function prologue.
6     ; set the GP:
7             lui     $28,%hi(__gnu_local_gp)
8             addiu     $sp,$sp,-32
9             addiu     $28,$28,%lo(__gnu_local_gp)
10     ; save the RA to the local stack:
11             sw         $31,28($sp)
12     ; load the address of the puts() function from the GP to
$25:
13             lw         $25,%call16(puts)($28)
14     ; load the address of the text string to $4 ($a0):
15             lui     $4,%hi($LC0)
16     ; jump to puts(), saving the return address in the link r
egister:
17             jalr     $25
18             addiu $4,$4,%lo($LC0)     ; branch delay slot
19     ; restore the RA:
20             lw         $31,28($sp)
21     ; copy 0 from $zero to $v0:
22             move     $2,$0
23     ; return by jumping to the RA:
24             j         $31
25     ; function epilogue:
26             addiu     $sp,$sp,32         ; branch delay slot
```

代码清单 3.19: 带优化的 GCC 4.4.5 (IDA)

```
1     .text:00000000     main:
2     .text:00000000
3     .text:00000000     var_10         = -0x10
4     .text:00000000     var_4         = -4
5     .text:00000000
6     ; function prologue.
7     ; set the GP:
8     .text:00000000                lui     $gp, (__gnu_local_g
p >> 16)
9     .text:00000004                addiu     $sp, -0x20
10     .text:00000008                la         $gp, (__gnu_loc
al_gp & 0xFFFF)
11     ; save the RA to the local stack:
12     .text:0000000C                sw         $ra, 0x20+var_4
($sp)
13     ; save the GP to the local stack:
14     ; for some reason, this instruction is missing in the GCC
 assembly output:
15     .text:00000010                sw         $gp, 0x20+var_1
0($sp)
16     ; load the address of the puts() function from the GP to
$t9:
17     .text:00000014                lw         $t9, (puts & 0x
FFFF)($gp)
18     ; form the address of the text string in $a0:
19     .text:00000018                lui     $a0, ($LC0 >> 16)
         # "Hello, world!"
20     ; jump to puts(), saving the return address in the link r
egister:
21     .text:0000001C                jalr     $t9
22     .text:00000020                la         $a0, ($LC0 & 0x
FFFF)     # "Hello, world!"
23     ; restore the RA:
24     .text:00000024                lw        $ra, 0x20+var_4(
$sp)
25     ; copy 0 from $zero to $v0:
26     .text:00000028                move     $v0, $zero
27     ; return by jumping to the RA:
28     .text:0000002C                jr         $ra
29     ; function epilogue:
30     .text:00000030                addiu     $sp, 0x20
```

3.5.3 无优化的 GCC

无优化的GCC会产生更冗长的代码：

代码清单 3.20: 无优化的 GCC 4.4.5 (汇编输出)

```
1     $LC0:
2             .ascii "Hello, world!\012\000"
3     main:
4     ; function prologue.
5     ; save the RA ($31) and FP in the stack:
6             addiu     $sp,$sp,-32
7             sw         $31,28($sp)
8             sw         $fp,24($sp)
9     ; set the FP (stack frame pointer):
10             move     $fp,$sp
11     ; set the GP:
12             lui     $28,%hi(__gnu_local_gp)
13             addiu     $28,$28,%lo(__gnu_local_gp)
14     ; load the address of the text string:
15             lui     $2,%hi($LC0)
16             addiu     $4,$2,%lo($LC0)
17     ; load the address of puts() using the GP:
18             lw         $2,%call16(puts)($28)
19             nop
20     ; call puts():
21             move     $25,$2
22             jalr     $25
23             nop              ; branch delay slot
24
25     ; restore the GP from the local stack:
26             lw         $28,16($fp)
27     ; set register $2 ($V0) to zero:
28             move     $2,$0
29     ; function epilogue.
30     ; restore the SP:
31             move     $sp,$fp
32     ; restore the RA:
33             lw         $31,28($sp)
34     ; restore the FP:
35             lw         $fp,24($sp)
36             addiu     $sp,$sp,32
37     ; jump to the RA:
38             j         $31
39             nop              ; branch delay slot
```

代码清单 3.21: 无优化的 GCC 4.4.5 (IDA)

```
1     .text:00000000     main:
2     .text:00000000
3     .text:00000000     var_10             = -0x10
4     .text:00000000     var_8             = -8
5     .text:00000000     var_4             = -4
6    .text:00000000
7     ; function prologue.
8     ; save the RA and FP in the stack:
9     .text:00000000                     addiu     $sp, -0x20
```

```
10      .text:00000004                          sw          $ra, 0x20+v
ar_4($sp)
11      .text:00000008                          sw          $fp, 0x20+v
ar_8($sp)
12      ; set the FP (stack frame pointer):
13     .text:0000000C move $fp, $sp
14      ; set the GP:
15      .text:00000010                          la          $gp, __gnu_
local_gp
16      .text:00000018                          sw          $gp, 0x20+v
ar_10($sp)
17      ; load the address of the text string:
18      .text:0000001C                          lui      $v0, (aHelloWo
rld >> 16)             # "Hello, world!"
19      .text:00000020                          addiu     $a0, $v0, (a
HelloWorld & 0xFFFF)     # "Hello, world!"
20      ; load the address of puts() using the GP:
21      .text:00000024                          lw          $v0, (puts
& 0xFFFF)($gp)
22      .text:00000028                          or          $at, $zero
    ; NOP
23      ; call puts():
24      .text:0000002C                          move     $t9, $v0
25      .text:00000030                          jalr     $t9
26     .text:00000034                          or          $at, $zero ;
 NOP
27      ; restore the GP from local stack:
28      .text:00000038                          lw          $gp, 0x20+v
ar_10($fp)
29      ; set register $2 ($V0) to zero:
30      .text:0000003C                          move     $v0, $zero
31      ; function epilogue.
32      ; restore the SP:
33      .text:00000040                          move     $sp, $fp
34      ; restore the RA:
35      .text:00000044                          lw          $ra, 0x20+v
ar_4($sp)
36      ; restore the FP:
37      .text:00000048                          lw          $fp, 0x20+v
ar_8($sp)
38      .text:0000004C                          addiu     $sp, 0x20
39      ; jump to the RA:
40      .text:00000050                          jr          $ra
41      .text:00000054                          or          $at, $zero
; NOP
```

3.5.4 堆栈结构在本例里面的作用

文本字符串的地址是通过寄存器传递的。那为什么要设置一个局部堆栈呢？这样做的原因是寄存器 `RA` 和 `GP` 的值必须被储存在某个地方(因为 `printf()` 被调用了)，局部堆栈就是用于这个目的的。如果这是个末端函数，那么有可能除去他的函

数开始和函数结尾，例如:2.3

3.5.5 带优化的 GCC:把它加载到GDB

代码清单 3.22: GDB session 的例子

```
root@debian-mips:~# gcc hw.c -O3 -o hw
root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licen
ses/gpl.html>
This is free software: you are free to change and redistribute i
t.
There is NO WARRANTY, to the extent permitted by law. Type "show
 copying"
and "show warranty" for details.
This GDB was configured as "mips-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/hw...(no debugging symbols found)...d
one.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw
Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:     lui     gp,0x42
0x00400644 <main+4>:     addiu     sp,sp,-32
0x00400648 <main+8>:     addiu     gp,gp,-30624
0x0040064c <main+12>:     sw         ra,28(sp)
0x00400650 <main+16>:     sw         gp,16(sp)
0x00400654 <main+20>:     lw         t9,-32716(gp)
0x00400658 <main+24>:     lui     a0,0x40
0x0040065c <main+28>:     jalr     t9
0x00400660 <main+32>:     addiu     a0,a0,2080
0x00400664 <main+36>:     lw         ra,28(sp)
0x00400668 <main+40>:     move     v0,zero
0x0040066c <main+44>:     jr         ra
0x00400670 <main+48>:     addiu     sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820: "hello, world"
(gdb)
```

3.5.5 小结

x86/ARM 和 x64/ARM64 代码的主要区别是：x64中指向字符串的指针是64位长度的。现代CPU是64位的主要原因是:内存成本的下降和各种应用对64位的强烈需求。我们现在能够给电脑加很多内存，以至于远远超过了32位指针能够寻址的范围。正因如此，现在所有的指针都是64位的了。

3.7 练习

- http://challenges.re/48
- http://challenges.re/49

第四章

函数的序幕和清尾

C语言的函数通常使用类似下面的代码片段作为序幕:

```
push  ebp
mov   ebp, esp
sub   esp, X
```

这些指令将EBP寄存器的值入栈，然后把ESP赋值给EBP，最后在栈中为局部变量分配一段空间。

在函数执行过程中EBP是固定的，可以用来作为访问局部变量和函数参数的基址。虽然也可以使用ESP，但在函数运行过程中，ESP可能会变化，使用起来不太方便。

函数清尾主要包括：释放栈中分配的空间，恢复EBP寄存器中的值，最后把控制流交由调用者：

```
mov  esp, ebp
pop  ebp
ret  0
```

函数的序幕和清尾代码片段通常被反汇编器作为函数定义的检测代码。

4.1 递归

函数的序幕和清尾代码片段可能会影响到递归函数的性能。

更多的信息请查看36.3一章。

第五章

栈

栈是计算机科学中最基本的一种数据结构。

从技术上讲,栈只是在**x86**中被 `ESP` 寄存器、**x64**中被 `RSP` 寄存器、或**ARM**中被 `SP` 寄存器指向的一块程序内存。

在x86和ARM Thumb模式中，访问栈最常用的指令是 `PUSH` 和 `POP` 。 `PUSH` 指令在32位模式下,会将 `ESP/RSP/SP` 的值减去4(在64位系统中,会减去8),然后将它唯一的参数写入到 `ESP/RSP/SP` 指向的内存地址。

`POP` 是 `PUSH` 的逆向操作:从 `SP` 指向的内存地址中获取数据,然后存入指定的参数中(一般为寄存器), 然后将栈指针加4(或8)。

在栈分配过后，栈指针指向栈底。 `PUSH` 减少栈指针； `POP` 指令增加栈指针。栈底实际上是栈分配到的内存的起始地址。这看起来很奇怪，但事实就是这样。

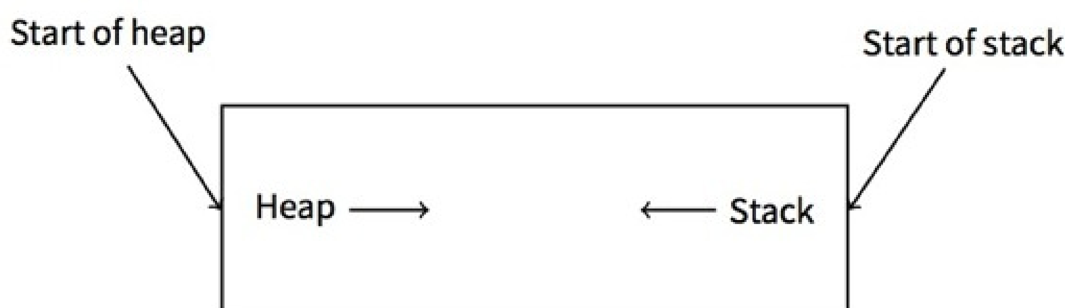
ARM 支持递增堆栈和递减堆栈。

举几个例子： `STMFD/LDMFD` ， `STMED/LDMED` 指令是用来处理递减堆栈的(向下增长，从高址开始向低址增长)。 `STMFA/LDMFA` ， `STMEA/LDMEA` 指令是用来处理递增堆栈的(向上增长，从低址开始向高址增长)。

5.1 为什么栈会反向增长?

从直觉上来说,我们会认为栈像其它数据结构一样,是向高地址正向增长的。

栈反向增长是有历史原因的。在计算机十分巨大，需要占据整个房间的年代，人们可以很容易的把内存分为两部分，一部分给堆，另一部分给栈。当然，在程序运行期间，我们并不知道堆栈各需要多大的空间。这时最简单的解决方法可能是：



在[RT74]中我们可以看到:

> 映像文件的用户核心部分可以被划分为三个逻辑段。程序代码段在虚拟内存里从0位置开始。 在程序运行过程中,这部分是具有写保护的,同一程序的所有进程都共享代码段的一个副本。 在虚拟内存地址中,程序代码段开始的8k字节，是私有的的可写数据段,这个段的大小可以通过系统调用来扩大。 从虚拟内存的高位地址开始是堆栈段，这部分像硬件栈指针大小的变动一样，可以自动的向下增长。

以上可以使我们联想到：一些学生在一个笔记本中写两门课的笔记：将第一门课的笔记正常写下，由于厌恶，而把另一门的笔记从后往前写。两种笔记有可能因缺少空间，而在中间的某处相遇。

5.2 栈可以用来做什么?

5.2.1 保存函数的返回地址

x86

当使用`CALL`指令去调用一个函数时,`CALL`后面一条指令的地址会被保存到栈中,然后使用无条件跳转指令跳转到`CALL`中执行。

`CALL`指令等价于`PUSH address_after_call / JMP operand`这对指令。

`RET`指令从栈中取出一个值并转跳到这个值上，这等价于`POP tmp / JMP tmp`指令对。

栈溢出是很容易的。只需要执行无止尽的递归。

```
void f()
{
        f();
};
```

MSVC 2008报告了这个问题:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.210
22.08 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: ’f’ : recursive on all contro
l paths, function will cause ⤦
    Ç runtime stack overflow
```

...但它还是生成了正确的代码:

```
?f@@YAXXZ PROC                          ; f
; File c:\tmp6\ss.cpp
; Line 2
        push     ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                          ; f
```

...如果我们设置优化标识(`/0x`),优化过的代码将不会出现栈溢出,并且会正确的运行的。（此处为反讽）

```
?f@@YAXXZ PROC                          ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                            ; f
```

GCC 4.4.1 在这两种条件下,会生成同样的代码,并且不会有任何警告。

ARM

ARM程序员也使用栈来保存返回地址,但稍有不同。正如我们在“Hello,World!(3.4)里提到过的, `RA` 被保存在 `LR` (链接寄存器)中。然而,如果有时需要调用另外一个函数,并且要多次使用 `LR` 寄存器,它的值必须被保存起来。通常它会在被保存到函数的开头。像我们经常看到的指令 `PUSH R4-R7, LR` ,与在函数结尾处的指令 `POP R4-R7, PC` ,在函数中使用到的寄存器的值，包括 `LR` ，会被保存到栈中。

然而,如果一个函数从未调用其它函数,它在 `RISC` 术语中被叫作叶子函数。因此,叶子函数不需要保存 `LR` 寄存器(因为他们并不修改它)。如果这样的函数很小，并只使用了少量的寄存器,它可能完全不需要用到栈。因此,可以不使用栈而调用叶子函数。这样做比在老x86机器上运行要快，因为不需要为栈留出额外的内存。在留给栈的内存尚未分配或不可用的情况下,这种方式是非常有用的。

一些叶子函数的例子：8.3.2 , 8.3.3 , 19.17 , 19.33 , 19.5.4 , 15.4 , 15.2 , 17.3 。

5.2.2 传递函数参数

在x86中,最常见的传参方式是 `cdecl` :

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

被调用函数通过栈指针得到参数。

因此,以下就是在函数f()的第一条指令执行之前，栈中参数的值是如何排列的。

ESP	return address
ESP+4	argument#1, marked in IDA as arg_0
ESP+8	argument#2, marked in IDA as arg_4
ESP+0xC	argument#3, marked in IDA as arg_8
...	...

关于对调用约定参见(64)。值得注意的是，没有任何东西强迫程序员一定要使用栈来传递参数。这并不是必需的,一个程序员完全可以不使用栈,而通过其它方式来实现参数传递。

例如,可以为参数分配一部分堆空间,存入参数,然后通过 `EAX` 寄存器里指向这个块的指针，将参数传递给函数。这样是可行的。然而,在**x86**和**ARM**中,使用栈传递参数还是更加方便的。

另外,被调函数并不知道有多少参数被传递进来。C语言中有些函数可以传递不同个数的参数(如: `printf()`),他们一般通过使用格式字符串(以 `%` 开始)来判断参数个数。

如果我们可以这样些： `printf("%d %d %d", 1234);`

printf()会输出入1234,然后另外输出和栈相邻的,两个另外的随机数字。

这就是为什么我们如何声明 `main()` 函数是不重要的,
像 `main()` , `main(int argc, char *argv[])` 或 `main(int argc, char *argv`
。

事实上, `CRT 模式` 大致是这样调用main()函数的：

```
push envp
push argv
push argc
call main
...
```

即使你将 `main()` 声明为不带参数的main()函数。它们仍然在栈中,只是没被使用。如果你将 `main()` 声明为 `main(int argc, char *argv[])`,你就可以使用前两个参数,并且第三个参数在你的函数仍然是"不可见的"。还有，如果你声明为 `main(int argc)` 这样,它同样是可以正常运行的。

5.2.3 存放局部变量

一个函数可以在栈中分配空间，用于储存局部变量。这只需要将栈指针向栈底增加。因此，无论你需要定义多少局部变量，这样都很快。

在栈中存放局部变量并不是一个硬性的要求。你可以将局部变量存到任何你想存的地方,但从传统上来说,大家更喜欢这样做。

5.2.4 x86: alloca() 函数

这里值得注意的是 `alloca()` 函数。该函数的作用类似于 `malloc()`,但它会直接在栈上分配内存。

它分配的内存块，并不需要调用像 `free()` 这样的函数来释放(4)。当函数运行结束, `ESP` 的值还原时,这部分内存会自动释放。

值得注意的是 `alloca()` 函数的实现。简而言之,这个函数就是根据你所需要的内存大小，将 `ESP` 指针指向栈底移位,然后将 `ESP` 指向所分配的内存块。

让我们试一下:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

`_snprintf()` 函数作用与 `printf()` 函数基本相同,不同的地方是 `printf()` 会将结果输出到的标准输出中(例如：终端和控制台), `_snprintf()` 会将结果保存到缓冲区中, `puts()` 会将缓冲区的内容复制到标准输出。当然，后面两行代码可以

使只用一个 `printf()` 调用替换,但我们必须说明小缓冲区的用途。

MSVC

让我们来编译 (MSVC 2010):

代码清单 5.1: MSVC 2010

```
...

        mov     eax, 600          ; 00000258H
        call    __alloca_probe_16
        mov     esi, esp

        push    3
        push    2
        push    1
        push    OFFSET $SG2672
        push    600               ; 00000258H
        push    esi
        call    __snprintf

        push    esi
        call    _puts
        add     esp, 28           ; 0000001cH

...
```

`alloca()` 的唯一一个参数通过 `EAX` 来传递(而不是把他压入栈)。在函数调用结束时, `ESP` 会指向 600字节的内存,我们可以像使用缓冲区数组一样来使用它。

GCC + Intel格式

GCC 4.4.1不需要调用额外的函数，就可以实现相同的功能:

代码清单 5.2: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push     ebp
        mov     ebp, esp
        push     ebx
        sub     esp, 660
        lea     ebx, [esp+39]
        and     ebx, -16                            ; align poi
nter by 16-bit border
        mov     DWORD PTR [esp], ebx                ; s
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %
d, %d\n"
        mov     DWORD PTR [esp+4], 600                ; maxlen
        call     _snprintf
        mov     DWORD PTR [esp], ebx                ; s
        call     puts
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC + AT&T 语法

我们来看看使用了AT&T语法的相同的代码:

代码清单 5.3: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        pushl     %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $660, %esp
        leal    39(%esp), %ebx
        andl    $-16, %ebx
        movl    %ebx, (%esp)
        movl    $3, 20(%esp)
        movl    $2, 16(%esp)
        movl    $1, 12(%esp)
        movl    $.LC0, 8(%esp)
        movl    $600, 4(%esp)
        call    _snprintf
        movl    %ebx, (%esp)
        call    puts
        movl    -4(%ebp), %ebx
        leave
        ret
```

这里的代码与上面的那个代码清单是相同的。

另外:在Intel语法中， `movl $3, 20(%esp)` 与 `mov DWORD PTR [esp + 20],3` 是等价的。在AT&T语法中，register+offset形式的内存地址表示为： `offset(%register)` 。

5.2.5 (Windows) 结构化异常处理 (SEH)

(如果存在) `SEH` 记录也是存放在栈中的。 想了解更多，参看(68.3)。

5.2.6 缓冲区溢出保护

想了解更多，参看(18.2)。

5.2.7 栈内数据的自动回收

也许把临时变量和 `SHE` 记录存在栈中，是因为他们会在函数的结尾会被自动的释放，而且只需要用一条指令就能还原栈指针(通常是 `ADD`)。函数的参数也会在函数的结尾被释放。相对的，储存在堆中的任何东西都必须被明确的释放。

5.3 典型的堆栈布局

以下是32位的环境中，第一个函数开始执行前，栈典型的布局:

...	...
ESP-0xC	local variable #2, marked in IDA as var_8
ESP-8	local variable #1, marked in IDA as var_4
ESP-4	saved value of EBP
ESP	return address
ESP+4	argument#1, marked in IDA as arg_0
ESP+8	argument#2, marked in IDA as arg_4
ESP+0xC	argument#3, marked in IDA as arg_8
...	...

5.4 栈内的'噪声'

在这本书里，我们经常提到栈中的"噪声"值和内存中的"垃圾"值，他们从哪里来？他们通常是上一个函数执行完而留下的值。简短的例子：

```
#include <stdio.h>

void f1()
{
        int a=1, b=2, c=3;
};
void f2()
{
        int a, b, c;
        printf ("%d, %d, %d\n", a, b, c);
};
int main()
{
        f1();
        f2();
};
```

编译后：

代码清单 5.4: 无优化的 MSVC 2010

```
$SG2752 DB         '%d, %d, %d', 0aH, 00H

_c$ = -12          ; size = 4
_b$ = -8          ; size = 4
_a$ = -4          ; size = 4
_f1     PROC
        push     ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     esp, ebp
        pop     ebp
        ret     0
_f1     ENDP

_c$ = -12          ; size = 4
_b$ = -8          ; size = 4
_a$ = -4          ; size = 4
_f2     PROC
        push     ebp
        mov     ebp, esp
        sub     esp, 12
        mov     eax, DWORD PTR _c$[ebp]
        push     eax
        mov     ecx, DWORD PTR _b$[ebp]
        push     ecx
        mov     edx, DWORD PTR _a$[ebp]
        push     edx
        push     OFFSET $SG2752 ; '%d, %d, %d'
        call     DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2 ENDP

_main     PROC
        push     ebp
        mov     ebp, esp
        call    _f1
        call     _f2
        xor     eax, eax
        pop     ebp
        ret     0
_main ENDP
```

编译器会有一些小怨言：

```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.402
19.01 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local varia
ble 'c' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local varia
ble 'b' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local varia
ble 'a' used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:st.exe
st.obj
```

但当我们运行编译好的程序时：

```
c:\Polygon\c>st
1, 2, 3
```

啊！这太奇怪了！在 `f2()` 里，我们并没有为任何变量赋值。这些幽灵般的值仍留在栈里。

让我们在OllyDbg里加载这个例子：

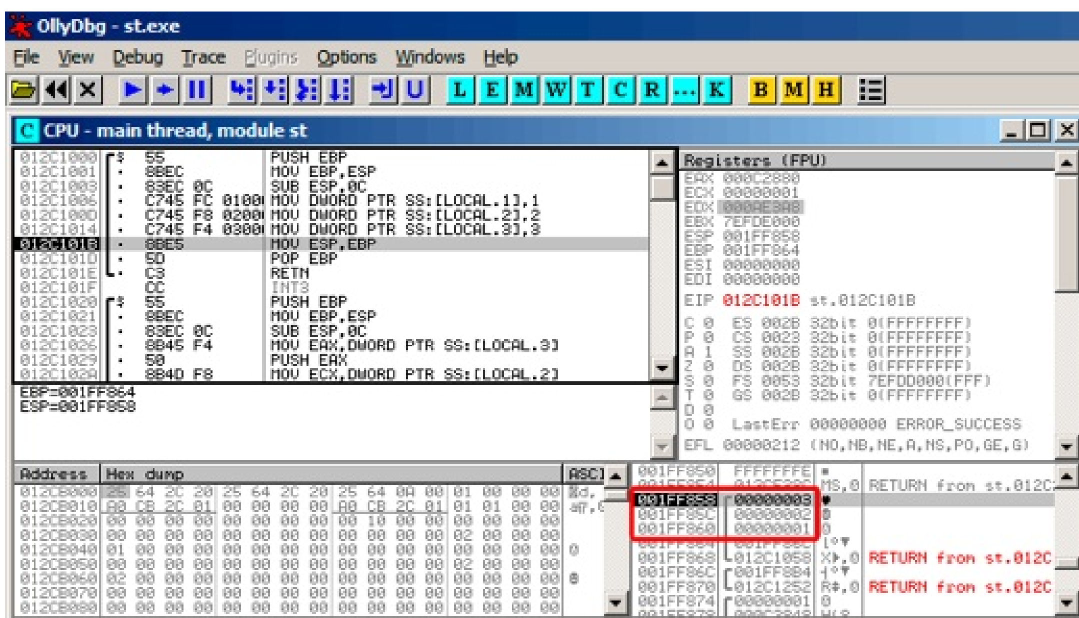


图 5.1: OllyDbg: f1()

当 `f1()` 分配变量a、b和c时，他们的值被存到 `0x1FF860` 等几个地址里。

然后当 `f2()` 执行的时候：

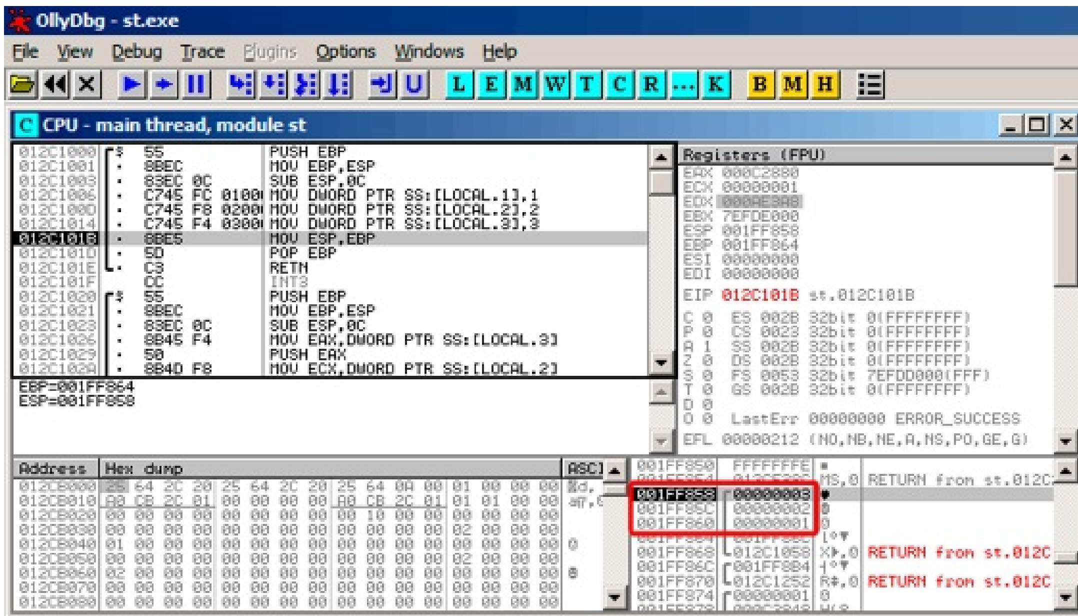


图 5.2: OllyDbg: f2()

... `f2()` 中的a、b和c分到了相同的地址！并且没有一个值被覆盖了，到目前为止他们的值未受影响。

为了让这中情况发生，一定有几个函数被依次调用，并且在每个函数分支中 `SP` 都有相同的值(例如，他们都有相同的参数)。然后这些临时变量就会被分配到栈中相同的位置上。

总的来说，栈中(和内存中)所有的值中总有几个，是先前的函数执行后留下的。严格的来说他们并不是随机的，但是他们的值是不可预测的。

还有其他可能吗？也许可以在每个函数执行完后，清除栈中一部分的值，但这会产生很多额外的(而且没必要的)工作。

5.4.1 MSVC 2013

这个例子是在MSVC 2010里编译的。但有些读者会尝试在MSVC 2013 里编译、运行它。然后会得到三个数字颠倒后的结果：

```
c:\Polygon\c>st
3, 2, 1
```

为什么？

我也在MSVC 2013中编译了这个程序：

代码清单 5.5: MSVC 2013

```
_a$ = -12           ; size = 4
_b$ = -8           ; size = 4
_c$ = -4           ; size = 4
_f2 PROC

...

_f2 ENDP

_c$ = -12           ; size = 4
_b$ = -8           ; size = 4
_a$ = -4           ; size = 4
_f1 PROC

...

_f1 ENDP
```

不像MSVC 2010 ，在MSVC 2013中 `f2()` 中的变量 `a/b/c` 会被以相反的顺序分配空间。但这样做是完全正确的，因为C/C++标准里并没有规定要以何种顺序来分配栈中的变量。产生这个区别的原因是：MSVC 2010有他自己的分配变量的方式，而在MSVC 2013中，可能有什么事改变了编译器内在的东西，所以结果稍有区别。

5.5 Exercises

- http://challenges.re/51
- http://challenges.re/52

第六章

printf() 与参数处理

现在让我们扩展"hello, world"(2)中的示例，将其中main()函数中printf的部分替换成这样

```
#include <stdio.h>
int main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
    return 0;
};
```

6.1 x86

6.1.1 x86: 3个参数

MSVC

在我们用MSVC 2010 Express编译后可以看到：

```
$SG3830 DB 'a=%d; b=%d; c=%d', 00H
...
        push 3
        push 2
        push 1
        push OFFSET $SG3830
        call _printf
        add esp, 16        ; 00000010H
```

这和之前的代码几乎一样，但是我们现在可以看到printf()的参数被反序压入了栈中。第一个参数被最后压入。

另外，在32bit的环境下int类型变量占4 bytes。

那么，这里有4个参数 4*4=16 —— 恰好在栈中占据了16bytes：一个32bit字符串指针，和3个int类型变量。

当函数执行完后，执行 `"ADD ESP, X"` 指令恢复栈指针寄存器(ESP 寄存器)。通常可以在这里推断函数参数的个数:用 X除以4。

当然，这只涉及__cdecl函数调用方式。

也可以在最后一个函数调用后，把几个`ADD ESP, X`指令合并成一个。

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

MSVC 与 ollyDbg

现在我们来在OllyDbg中加载这个范例。我们可以尝试在MSVC 2012 加 /MD 参数编译这个示例，也就是链接`MSVCR*.dll`，那么我们就可以在debugger中清楚的看到调用的函数。

在OllyDbg中载入程序，最开始的断点在ntdll.dll中，接着按F9(run)，然后第二个断点在CRT-code中。现在我们来找main()函数。

往下滚动屏幕，找到下图这段代码(MSVC把main()函数分配在代码段开始处) 见图5.3

点击 PUSH EBP指令，按下F2(设置断点)然后按下F9(run),通过这些操作来跳过CRT-code，因为我们现在还不必关注这部分。

按6次F8(step over)。见图5.4 现在EIP 指向了CALL printf的指令。和其他调试器一样，OllyDbg高亮了有值改变的寄存器。所以每次你按下F8,EIP都在改变然后它看起来便是红色的。ESP同时也在改变，因为它是指向栈的

栈中的数据又在哪？那么看一下调试器右下方的窗口:

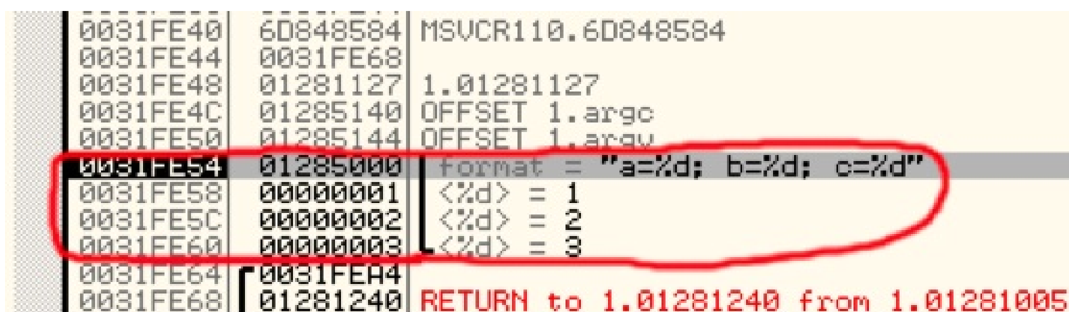


图 6.1

然后我们可以看到有三列，栈的地址，元组数据，以及一些OllyDbg的注释，OllyDbg可以识别像printf()这样的字符串，以及后面的三个值。

右击选中字符串，然后点击"follow in dump",然后字符串就会出现在左侧显示内存数据的地方，这些内存的数据可以被编辑。我们可以修改这些字符串，之后这个例子的结果就会变的不同，现在可能并不是很实用。但是作为练习却非常好，可以体会每部分是如何工作的。

再按一次F8(step over)

然后我们就可以看到输出

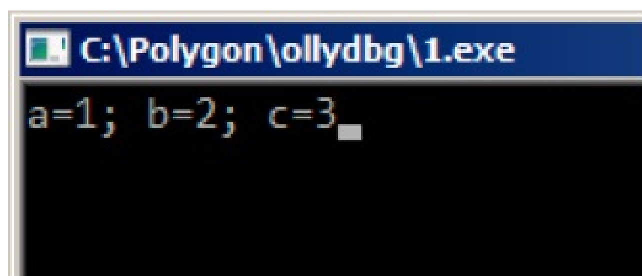


图6.2 执行printf()函数

让我们看看寄存器和栈是怎样变化的 见图5.5

EAX寄存器现在是0xD(13).这是正确的，printf()返回打印的字符，EIP也变了——

事实上现在指向CALL printf之后下一条指令的地址.ECX和EDX的值也改变了。显然，printf()函数的内部机制对它们进行了使用。

很重要的一点ESP的值并没有发生变化，栈的状态也是！我们可以清楚地看到字符串和相应的3个值还是在那里，实际上这就是cdecl调用方式。被调用的函数并不清楚栈中参数，因为这是调用体的任务。

再按一下F8执行 `ADD ESP, 0` 见图5.6

ESP改变了，但是值还是在栈中！当然 没有必要用0或者别的数据填充这些值。

因为在栈指针寄存器之上的数据都是无用的。

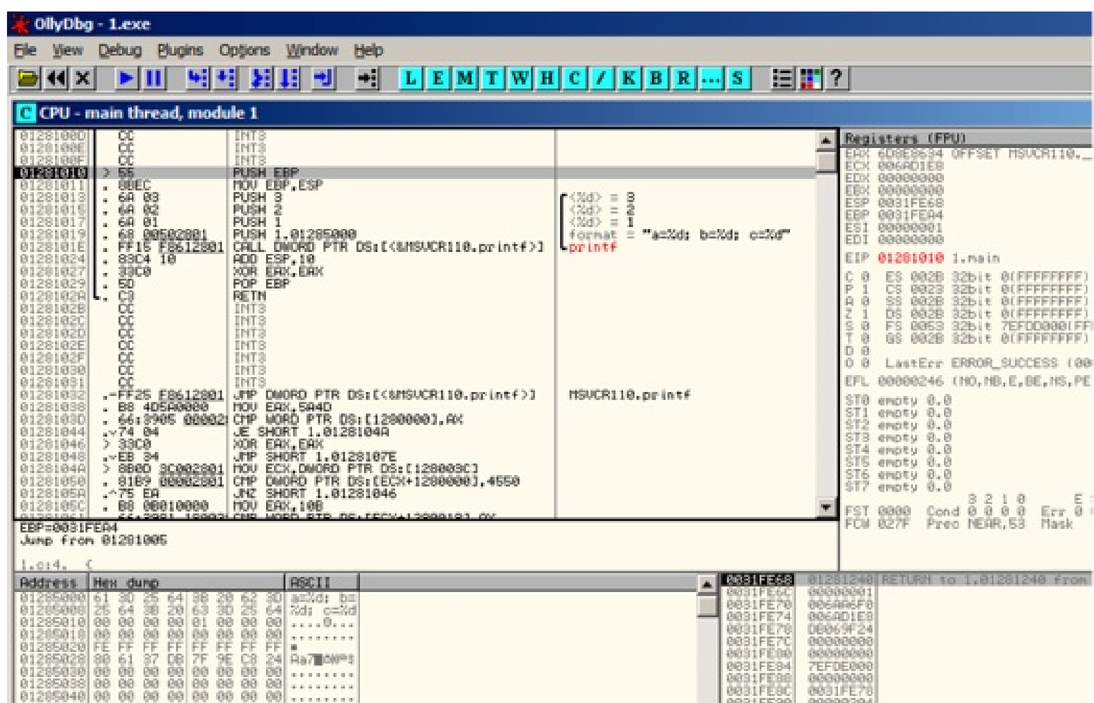


图6.3 OllyDbg:main()初始处

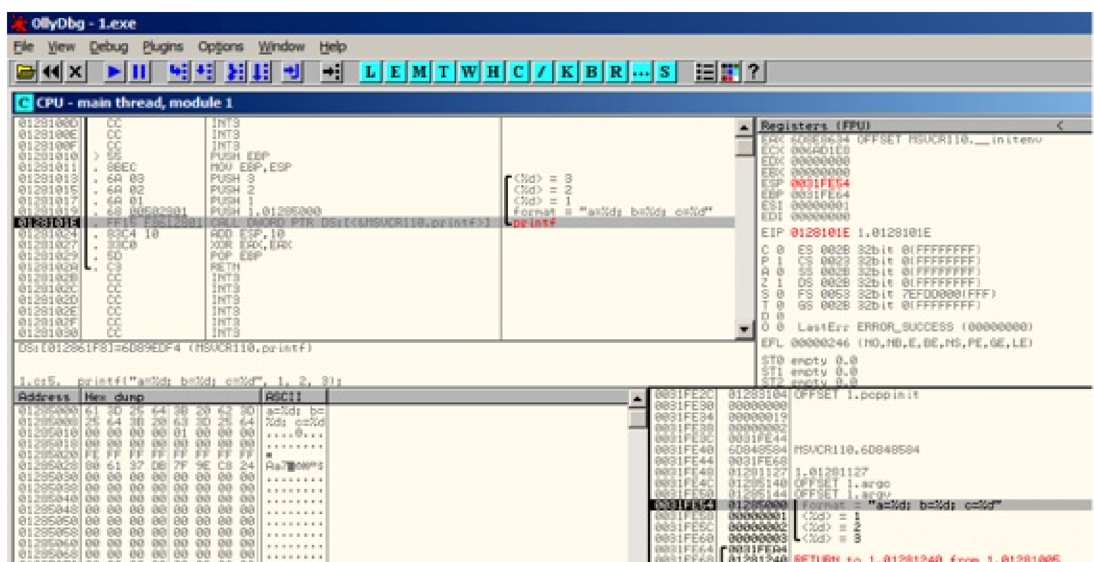


图6.4 OllyDbg:printf()执行时



图6.5 Ollydbg:printf()执行后



图6.6 OllyDbg ADD ESP, 10执行完后

GCC

现在我们将同样的程序在linux下用GCC4.4.1编译后放入IDA看一下:

```
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
                mov     [esp+10h+var_4], 3
                mov     [esp+10h+var_8], 2
                mov     [esp+10h+var_C], 1
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

MSVC与GCC编译后代码的不同点只是参数入栈的方法不同，这里GCC不用PUSH/POP而是直接对栈操作。

GCC与GDB

接着我们尝试在linux中用GDB运行下这个示例程序。

-g 表示将debug信息插入可执行文件中

```
$ gcc 1.c -g -o 1
```

反编译：

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licen
ses/gpl.html>
This is free software: you are free to change and redistribute i
t.
There is NO WARRANTY, to the extent permitted by law. Type "show
 copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/1...done.
```

表6.1 在printf()处设置断点

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Run 这里没有printf()函数的源码，所以GDB没法显示出源码，但是却可以这样做

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at
printf.c:29
29 printf.c: No such file or directory.
```

打印10组栈中的元组数据，左边是栈中的地址

```
(gdb) x/10w $esp
0xbffff11c: 0x0804844a 0x080484f0 0x00000001 0x00000002
0xbffff12c: 0x00000003 0x08048460 0x00000000 0x00000000
0xbffff13c: 0xb7e29905 0x00000001
```

最开始的是返回地址(0x0804844a),我们可以确定在这里，于是可以反汇编这里的代码

```
(gdb) x/5i 0x0804844a
0x804844a <main+45>: mov $0x0,%eax
0x804844f <main+50>: leave
0x8048450 <main+51>: ret
0x8048451: xchg %ax,%ax
0x8048453: xchg %ax,%ax
```

两个XCHG指令，明显是一些垃圾数据,可以忽略 第二个(0x080484f0)是一处格式化字符串

```
(gdb) x/s 0x080484f0
0x80484f0: "a=%d; b=%d; c=%d"
```

而其他三个则是printf()函数的参数，另外的可能只是栈中的垃圾数据，但是也可能是其他函数的数据，例如它们的本地变量。这里可以忽略。 执行 finish ，表示执行到函数结束。在这里是执行到printf()完。

```
(gdb) finish
Run till exit from #0 __printf (format=0x80484f0 "a=%d; b=%d; c=
%d") at printf.c:29
main () at 1.c:6
6 return 0;
Value returned is $2 = 13
```

GDB显示了printf()函数在eax中的返回值，这是打印字符的数量，就像在OllyDbg中一样。

我们同样看到了 `return 0;` 及这在1.c文件中第6行所代表的含义。1.c文件就在当前目录下，GDB就在那找到了字符串。但是GDB又是怎么知道当前执行到了哪一行？

事实上这和编译器有关，当生成调试信息时，同样也保存了一张代码行号与指令地址的关系表。

查看EAX中储存的13:

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
...
```

反汇编当前的指令

```
(gdb) disas
Dump of assembler code for function main:
    0x0804841d <+0>:    push   %ebp
    0x0804841e <+1>:    mov    %esp,%ebp
    0x08048420 <+3>:    and    $0xfffffff0,%esp
    0x08048423 <+6>:    sub    $0x10,%esp
    0x08048426 <+9>:    movl   $0x3,0xc(%esp)
    0x0804842e <+17>:   movl   $0x2,0x8(%esp)
    0x08048436 <+25>:   movl   $0x1,0x4(%esp)
    0x0804843e <+33>:   movl   $0x80484f0,(%esp)
    0x08048445 <+40>:   call   0x80482f0 <printf@plt>
=>  0x0804844a <+45>:   mov    $0x0,%eax
    0x0804844f <+50>:   leave
    0x08048450 <+51>:   ret
End of assembler dump.
```

GDB默认使用AT&T语法显示，当然也可以转换至intel:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
    0x0804841d <+0>:    push    ebp
    0x0804841e <+1>:    mov     ebp,esp
    0x08048420 <+3>:    and     esp,0xfffffff0
    0x08048423 <+6>:    sub     esp,0x10
    0x08048426 <+9>:    mov     DWORD PTR [esp+0xc],0x3
    0x0804842e <+17>:   mov     DWORD PTR [esp+0x8],0x2
    0x08048436 <+25>:   mov     DWORD PTR [esp+0x4],0x1
    0x0804843e <+33>:   mov     DWORD PTR [esp],0x80484f0
    0x08048445 <+40>:   call    0x80482f0 <printf@plt>
=>  0x0804844a <+45>:   mov     eax,0x0
    0x0804844f <+50>:   leave
    0x08048450 <+51>:   ret
End of assembler dump.
```

执行下一条指令,GDB显示了结束大括号，代表着这里是函数结束部分。

```
(gdb) step
7 };
```

在执行完 `MOV EAX, 0` 后我们可以看到EAX就已经变为0了。

```
(gdb) info registers
eax 0x0 0
ecx 0x0 0
edx 0x0 0
ebx 0xb7fc0000 -1208221696
esp 0xbffff120 0xbffff120
ebp 0xbffff138 0xbffff138
esi 0x0 0
edi 0x0 0
eip 0x804844f 0x804844f <main+50>
...
```

6.1.2 x64: 8个参数

为了看其他参数如何通过栈传递的，我们再次修改代码将参数个数增加到9个(printf()格式化字字符串和8个int 变量)

```
#include <stdio.h>
int main() {
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d
", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

MSVC

正如我们之前所见，在win64下开始的4个参数传递至RCX，RDX，R8，R9寄存器，

然而 MOV指令，替代PUSH指令。用来准备栈数据，所以值都是直接写入栈中

```
$SG2923 DB ’a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d’, 0aH
, 00H

main    PROC
        sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
        mov     edx, 1
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf

        ; return 0
        xor eax, eax

        add     rsp, 88
        ret     0
main ENDP
_TEXT ENDS
END
`
```

表6.2：msvc 2010 x64

GCC

在*NIX系统，对于x86-64这也是同样的原理，除了前6个参数传递给了RDI，RSI，RDX，RCX，R8，R9寄存器。GCC将生成的代码字符指针写入了EDI而不是RDI(如果有的话)——我们在2.2.2节看到过这部分

同样我们也看到在寄存器EAX被清零前有个`printf() call`：

```
.LC0:
    .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d
"

main:
    sub     rsp, 40

    mov     r9d, 5
    mov     r8d, 4
    mov     ecx, 3
    mov     edx, 2
    mov     esi, 1
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax ; number of vector registers passed
    mov     DWORD PTR [rsp+16], 8
    mov     DWORD PTR [rsp+8], 7
    mov     DWORD PTR [rsp], 6
    call    printf

    ; return 0

    xor     eax, eax
    add     rsp, 40
    ret
```

表6.3:GCC 4.4.6 –o 3 x64

GCC + GDB

让我们在GDB中尝试这个例子。

```
$ gcc -g 2.c -o 2
```

反编译：

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licen
ses/gpl.html>
This is free software: you are free to change and redistribute i
t.
There is NO WARRANTY, to the extent permitted by law. Type "show
 copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/2...done.
```

表5.4:在printf()处下断点，然后run

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2
Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d;
 e=%d; f=%d; g=%d; h=%d
") at
printf.c:29
29 printf.c: No such file or directory.
```

寄存器RSI/RDX/RCX/R8/R9都有应有的值，RIP则是printf()函数地址

```
(gdb) info registers
rax     0x0     0
rbx     0x0     0
rcx     0x3     3
rdx     0x2     2
rsi     0x1     1
rdi     0x400628 4195880
rbp     0x7fffffffdf60 0x7fffffffdf60
rsp     0x7fffffffdf38 0x7fffffffdf38
r8      0x4     4
r9      0x5     5
r10     0x7fffffffdce0 140737488346336
r11     0x7ffff7a65f60 140737348263776
r12     0x400440 4195392
r13     0x7fffffffe040 140737488347200
r14     0x0     0
r15     0x0     0
rip     0x7ffff7a65f60 0x7ffff7a65f60 <__printf>
...
```

表5.5 检查格式化字符串

```
(gdb) x/s $rdi
0x400628: "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d
"
```

用x/g命令显示栈内容

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576 0x0000000000000006
0x7fffffffdf48: 0x0000000000000007 0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000 0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5 0x0000000000000000
0x7fffffffdf78: 0x00007fffffffe048 0x0000000100000000
```

与之前一样，第一个栈元素是返回地址，我们也同时也看到在高32位的8也没有被清除。 0x00007fff00000008，这是因为是32位int类型的，因此，高寄存器或堆栈部分可能包含一些随机垃圾数值。

printf()函数执行之后将返回控制，GDB会显示整个main()函数。

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main:
    0x000000000040052d <+0>:    push    rbp
    0x000000000040052e <+1>:    mov     rbp,rsp
    0x0000000000400531 <+4>:    sub     rsp,0x20
    0x0000000000400535 <+8>:    mov     DWORD PTR [rsp+0x10],0x8
    0x000000000040053d <+16>:   mov     DWORD PTR [rsp+0x8],0x7
    0x0000000000400545 <+24>:   mov     DWORD PTR [rsp],0x6
    0x000000000040054c <+31>:   mov     r9d,0x5
    0x0000000000400552 <+37>:   mov     r8d,0x4
    0x0000000000400558 <+43>:   mov     ecx,0x3
    0x000000000040055d <+48>:   mov     edx,0x2
    0x0000000000400562 <+53>:   mov     esi,0x1
    0x0000000000400567 <+58>:   mov     edi,0x400628
    0x000000000040056c <+63>:   mov     eax,0x0
    0x0000000000400571 <+68>:   call    0x400410 <printf@plt>
    0x0000000000400576 <+73>:   mov     eax,0x0
    0x000000000040057b <+78>:   leave
    0x000000000040057c <+79>:   ret
End of assembler dump.
```

执行完printf()后，就会清零EAX，然后发现EAX早已为0，RIP现在则指向LEAVE指令。

```
(gdb) finish
Run till exit from #0 __printf (format=0x400628 "a=%d; b=%d; c=%
d; d=%d; e=%d; f=%d; g=%d; h=%d
n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6       return 0;
Value returned is $1 = 39
(gdb) next
7 };
(gdb) info registers
rax     0x0     0
rbx     0x0     0
rcx     0x26    38
rdx     0x7ffff7dd59f0 140737351866864
rsi     0x7fffffd9 2147483609
rdi     0x0     0
rbp     0x7fffffffdf60 0x7fffffffdf60
rsp     0x7fffffffdf40 0x7fffffffdf40
r8      0x7ffff7dd26a0 140737351853728
r9      0x7ffff7a60134 140737348239668
r10     0x7fffffffd5b0 140737488344496
r11     0x7ffff7a95900 140737348458752
r12     0x400440 4195392
r13     0x7fffffffe040 140737488347200
r14     0x0     0
r15     0x0     0
rip     0x40057b 0x40057b <main+78>
...
```

6.2 ARM

6.3 ARM:3个参数

习惯上，ARM传递参数的规则(参数调用)如下:前4个参数传递给了R0-R3寄存器，其余的参数则在栈中。这和fastcall或者win64传递参数很相似

32-bit ARM

Non-optimizing Keil + ARM mode(非优化keil编译模式 + ARM环境)

```
.text:00000014              printf_main1
.text:00000014 10 40 2D E9        STMFD   SP!, {R4,LR}
.text:00000018 03 30 A0 E3        MOV     R3, #3
.text:0000001C 02 20 A0 E3        MOV     R2, #2
.text:00000020 01 10 A0 E3        MOV     R1, #1
.text:00000024 1D 0E 8F E2        ADR     R0, aADBDCD ; "a=%d; b=
%d; c=%d
"
.text:00000028 0D 19 00 EB        BL      __2printf
.text:0000002C 10 80 BD E8        LDMFD   SP!, {R4,PC}
```

所以 前四个参数按照它们的顺序传递给了R0-R3， printf()中的格式化字符串指针在R0中，然后1在R1，2在R2，3在R3. 到目前为止没有什么不寻常的。

Optimizing Keil + ARM mode(优化的keil编译模式 + ARM环境)

```
.text:00000014     EXPORT printf_main1
.text:00000014     printf_main1
.text:00000014 03 30 A0 E3     MOV    R3, #3
.text:00000018 02 20 A0 E3     MOV    R2, #2
.text:0000001C 01 10 A0 E3     MOV    R1, #1
.text:00000020 1E 0E 8F E2     ADR    R0, aADBDCD ; "a=%d; b=%d;
 c=%d
"
.text:00000024 CB 18 00 EA     B     __2printf
```

表5.7: Optimizing Keil + ARM mode

这是在针对ARM optimized (-O3)版本下的，我们可以B作为最后一个指令而不是熟悉的BL。另外一个不同之处在optimized与之前的(compiled without optimization)对比发现函数prologue 和 epilogue(储存R0和LR值的寄存器)，B指令仅仅跳向另一处地址，没有任何关于LR寄存器的操作，也就是说它和x86中的jmp相似，为什么会这样？因为代码就是这样，事实上，这和前面相似，主要有两点原因 1)不管是栈还是SP(栈指针)，都有被修改。2)printf()的调用是最后的指令，所以之后便没有了。完成之后，printf()函数就返回到LR储存的地址处。但是指针地址从函数调用的地方转移到了LR中！接着就会从printf()到那里。结果，我们不需要保存LR，因为我们没有必要修改LR。因为除了printf()函数外没有其他函数了。另外，除了这个调用外，我们不需要再做别的。这就是为什么这样编译是可行的。

Optimizing Keil + thumb mode

```
.text:0000000C     printf_main1
.text:0000000C 10 B5           PUSH {R4,LR}
.text:0000000E 03 23           MOVS R3, #3
.text:00000010 02 22           MOVS R2, #2
.text:00000012 01 21           MOVS R1, #1
.text:00000014 A4 A0           ADR R0, aADBDCD ; "a=%d; b=%d; c=
%d
"
.text:00000016 06 F0 EB F8     BL __2printf
.text:0000001A 10 BD           POP {R4,PC}
```

表6.8：Optimizing Keil + thumb mode

和non-optimized for ARM mode代码没什么明显的区别

Optimizing Keil 6/2013 (ARM mode) + 让我们移除 return

ARM 64

Non-optimizing GCC (Linaro) 4.9

6.2.2 ARM: 8 arguments

我们再用之前9个参数的那个例子

```
void printf_main2()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d
", 1, 2, 3, 4, 5, 6, 7, 8);
};
```

Optimizing Keil: ARM mode

```
.text:00000028      printf_main2
.text:00000028
.text:00000028      var_18 = -0x18
.text:00000028      var_14 = -0x14
.text:00000028      var_4 = -4
.text:00000028
.text:00000028 04 E0 2D E5      STR     LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2      SUB     SP, SP, #0x14
.text:00000030 08 30 A0 E3      MOV     R3, #8
.text:00000034 07 20 A0 E3      MOV     R2, #7
.text:00000038 06 10 A0 E3      MOV     R1, #6
.text:0000003C 05 00 A0 E3      MOV     R0, #5
.text:00000040 04 C0 8D E2      ADD     R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8      STMIA   R12, {R0-R3}
.text:00000048 04 00 A0 E3      MOV     R0, #4
.text:0000004C 00 00 8D E5      STR     R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3      MOV     R3, #3
.text:00000054 02 20 A0 E3      MOV     R2, #2
.text:00000058 01 10 A0 E3      MOV     R1, #1
.text:0000005C 6E 0F 8F E2      ADR     R0, aADBDCDDDEDFDGD ; "a
=%d; b=%d; c=%d; d=%d;
e=%d; f=%d; g=%"...
.text:00000060 BC 18 00 EB      BL      __2printf
.text:00000064 14 D0 8D E2      ADD     SP, SP, #0x14
.text:00000068 04 F0 9D E4      LDR     PC, [SP+4+var_4],#4
```

这些代码可以分成几个部分:

Function prologue:

最开始的”STR LR, [SP,#var_4]!”指令将LR储存在栈中，因为我们将用这个寄存器调用printf()。

第二个” SUB SP, SP, #0x14”指令减了SP(栈指针),为了在栈上分配0x14(20)bytes的内存，实际上我们需要传递5个 32-bit的数据通过栈传递给printf()函数，而且每个占4bytes，也就是5*4=20。另外4个32-bit的数据将会传递给寄存器。

通过栈传递5，6，7和8：

然后，5，6，7，8分别被写入了R0，R1，R2及R3寄存器。然后 `ADD R12, SP,#0x18+var_14` 指令将栈中指针的地址写入，并且在这里会向R12写入4个值， `var_14` 是一个汇编宏，相当于0x14，这些都由IDA简明的创建表示访问栈的变量，var_?在IDA中表示栈中的本地变量，所以SP+4将被写入R12寄存器。下一步的”STMIA R12, R0-R3”指令将R0-R3寄存器的内容写在了R2指向的指针处。STMIA指令指Store Multiple Increment After, Increment After指R12寄存器在有值写入后自增4。

通过栈传递4:

4存在R0中，然后这个值在” STR R0, [SP,#0x18+var_18]”指令帮助下，存在了栈上，var_18是0x18，偏移量为0.所以R0寄存器中的值将会写在SP指针指向的指针处。

通过寄存器传递1，2，3:

开始3个数(a,b,c)(分别是1,2,3)正好在printf()函数调用前被传递到了R1，R2，R3寄存器中。 然后另外5个值通过栈传递。

printf() 调用:

“ADD SP, SP, #0x14”指令将SP指针返回到之前的指针处，因此清除了栈，当然，栈中之前写入的数据还在那，但是当后来的函数被调用时那里则会被重写。“LDR PC, [SP+4+var_4],#4"指令将LR中储存的值载入到PC指针，因此函数结束。

Optimizing Keil: thumb mode

```
.text:0000001C      printf_main2
.text:0000001C
.text:0000001C      var_18 = -0x18
.text:0000001C      var_14 = -0x14
.text:0000001C      var_8 = -8
.text:0000001C
.text:0000001C 00 B5        PUSH    {LR}
.text:0000001E 08 23        MOVS    R3, #8
.text:00000020 85 B0        SUB     SP, SP, #0x14
.text:00000022 04 93        STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22        MOVS    R2, #7
.text:00000026 06 21        MOVS    R1, #6
.text:00000028 05 20        MOVS    R0, #5
.text:0000002A 01 AB        ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3        STMIA   R3!, {R0-R2}
.text:0000002E 04 20        MOVS    R0, #4
.text:00000030 00 90        STR     R0, [SP,#0x18+var_18]
.text:00000032 03 23        MOVS    R3, #3
.text:00000034 02 22        MOVS    R2, #2
.text:00000036 01 21        MOVS    R1, #1
.text:00000038 A0 A0        ADR     R0, aADBDCDDDEDFDGD ; "a=%d;
 b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
.text:0000003A 06 F0 D9 F8  BL __2printf
.text:0000003E
.text:0000003E              loc_3E ; CODE XREF: example13_f+16
.text:0000003E 05 B0        ADD SP, SP, #0x14
.text:00000040 00 BD        POP {PC}
```

几乎和之前的例子是一样的，然后这是thumb 代码，值入栈的确不同:先是8，然后5，6，7，第三个是4。

5.4.3 Optimizing Xcode (LLVM): ARM mode

```
__text:0000290C     _printf_main2
__text:0000290C
__text:0000290C     var_1C = -0x1C
__text:0000290C     var_C = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9     STMFD   SP!, {R7,LR}
__text:00002910 0D 70 A0 E1     MOV     R7, SP
__text:00002914 14 D0 4D E2     SUB     SP, SP, #0x14
__text:00002918 70 05 01 E3     MOV     R0, #0x1570
__text:0000291C 07 C0 A0 E3     MOV     R12, #7
__text:00002920 00 00 40 E3     MOVT    R0, #0
__text:00002924 04 20 A0 E3     MOV     R2, #4
__text:00002928 00 00 8F E0     ADD     R0, PC, R0
__text:0000292C 06 30 A0 E3     MOV     R3, #6
__text:00002930 05 10 A0 E3     MOV     R1, #5
__text:00002934 00 20 8D E5     STR     R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9     STMFA   SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3     MOV     R9, #8
__text:00002940 01 10 A0 E3     MOV     R1, #1
__text:00002944 02 20 A0 E3     MOV     R2, #2
__text:00002948 03 30 A0 E3     MOV     R3, #3
__text:0000294C 10 90 8D E5     STR     R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB     BL      _printf
__text:00002954 07 D0 A0 E1     MOV     SP, R7
__text:00002958 80 80 BD E8     LDMFD   SP!, {R7,PC}
```

几乎和我们之前遇到的一样，除了STMFA(Store Multiple Full Ascending)指令，它和STMIB(Store Multiple Increment Before)指令一样，这个指令直到下个寄存器的值写入内存时会增加SP寄存器中的值，但是反过来却不同。

另外一个地方我们可以轻松的发现指令是随机分布的，例如，R0寄存器中的值在三个地方初始，在0x2918，0x2920,0x2928。而这一个指令就可以搞定。然而，optimizing compiler有它自己的原因，对于如何更好的放置指令，通常，处理器尝试同时执行并行的指令，例如像” MOVT R0, #0”和” ADD R0, PC,R0”就不能同时执行了，因为它们同时都在修改R0寄存器，另一方面”MOVT R0, #0”和”MOV R2, #4”指令却可以同时执行，因为执行效果并没有任何冲突。 大概，编译器就是这样尝试编译的，可能。

Optimizing Xcode (LLVM): thumb-2 mode

```
__text:00002BA0     _printf_main2
__text:00002BA0
__text:00002BA0     var_1C = -0x1C
__text:00002BA0     var_18 = -0x18
__text:00002BA0     var_C = -0xC
__text:00002BA0
__text:00002BA0 80 B5           PUSH    {R7,LR}
__text:00002BA2 6F 46           MOV     R7, SP
__text:00002BA4 85 B0           SUB     SP, SP, #0x14
__text:00002BA6 41 F2 D8 20     MOVW    R0, #0x12D8
__text:00002BAA 4F F0 07 0C     MOV.W   R12, #7
__text:00002BAE C0 F2 00 00     MOVT.W  R0, #0
__text:00002BB2 04 22           MOVS    R2, #4
__text:00002BB4 78 44           ADD     R0, PC ; char *
__text:00002BB6 06 23           MOVS    R3, #6
__text:00002BB8 05 21           MOVS    R1, #5
__text:00002BBA 0D F1 04 0E     ADD.W   LR, SP, #0x1C+var_18
__text:00002BBE 00 92           STR     R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09     MOV.W   R9, #8
__text:00002BC4 8E E8 0A 10     STMIA.W LR, {R1,R3,R12}
__text:00002BC8 01 21           MOVS    R1, #1
__text:00002BCA 02 22           MOVS    R2, #2
__text:00002BCC 03 23           MOVS    R3, #3
__text:00002BCE CD F8 10 90     STR.W   R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA     BLX     _printf
__text:00002BD6 05 B0           ADD     SP, SP, #0x14
__text:00002BD8 80 BD           POP     {R7,PC}
```

几乎和前面的例子相同，除了thumb-instructions在这里被替代使用了

ARM 64

无优化的 GCC (Linaro) 4.9

6.3 MIPS

6.3.1 3个参数

带优化的 GCC 4.4.5

无优化的 GCC 4.4.5

6.3.2 8个参数

带优化的 GCC 4.4.5

无优化的 GCC 4.4.5

6.4 结论

6.5 By the way

值得一提的是，这些x86,x64,fastcall和ARM传递参数的不同表现了CPU并不在意函数参数是怎样传递的，同样也假想编译器可能用特殊的结构传送参数而一点也不是通过栈。

第七章

scanf()

现在我们来使用scanf()。

7.1 简单的例子

```
#include <stdio.h>
int main()
{
    int x;
    printf ("Enter X:
");
    scanf ("%d", &x);
    printf ("You entered %d...
", x);
    return 0;
};
```

如今使用scanf()作为用户交互非常不明智，但是我们还是可以说明如何把指针传递给int变量。

7.1.1 关于指针

指针是计算机科学中最基础的概念之一。通常，大数组、结构或对象作为参数被传递给其它函数花费太大，而传递它们的地址要相对简单的多。此外：如果调用函数要修改作为参数传进来的数组或结构中的数据，并将其整体返回，那这种情况就太荒唐了。因此最简单的办法就是把数组或结构的地址传递给函数，让函数进行修改。

在C/C++中指针就是某处内存的地址。

在x86中，地址是以32位数表示的（占4字节）；在x86-64中是64位数（占8字节）。顺便一说，这也是为什么有些人在改用x86-64时感到愤怒——x64架构中所有的指针需要的空间是原来的两倍。

通过某种方法，只使用无类型指针也是可行的。例如标准C函数memcpy()，用于把一个区块复制到另外一个区块上，需要两个void*型指针作为输入，因为你无法预知，也无需知道要复制区块的类型，区块的大小才是重要的。

当函数需要一个以上的返回值时也经常用到指针（等到第十章再讲）。scanf()就是这样，函数除了要显示成功读入的字符个数外，还要返回全部值。

在C/C++中，指针类型只是用于在编译阶段进行类型检查。本质上，在已编译的代码中并不包含指针类型的信息。

7.1.2 x86

MSVC

MVSC 2010编译后得到下面代码

```
CONST SEGMENT
$SG3831 DB ’Enter X:’, 0aH, 00H
$SG3832 DB ’%d’, 00H
35
6.2. X86 CHAPTER 6. SCANF()
$SG3833 DB ’You entered %d...’, 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_x$ = -4 ; size = 4
_main PROC
        push    ebp
        mov     ebp, esp
        push    ecx
        push    OFFSET $SG3831 ; ’Enter X:’
        call    _printf
        add     esp, 4
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3832 ; ’%d’
        call    _scanf
        add     esp, 8
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3833 ; ’You entered %d...’
        call    _printf
        add     esp, 8
        ; return 0
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main ENDP
_TEXT ENDS
```

X是局部变量。

C/C++标准告诉我们它只对函数内部可见，无法从外部访问。习惯上，局部变量放在栈中。也可能有其他方法，但在x86中是这样。

函数序言后下一条指令PUSH ECX目的并不是要存储ECX的状态（注意程序结尾没有与之相对的POP ECX）。

事实上这条指令仅仅是在栈中分配了4字节用于存储变量x。

变量x可以用宏 _x$ 来访问（等于-4），EBP寄存器指向当前栈帧。

在一个函数执行时，EBP将指向当前栈帧，通过EBP+offset来访问局部变量和函数参数也是可行的。

也可以使用ESP寄存器达到相同目的，但由于它经常变化所以使用不方便。EBP值保存了进入函数时ESP的值。

下面是一个非常典型的32位栈帧结构

```
...
EBP-8    local variable #2, marked in IDA as var_8
EBP-4    local variable #1, marked in IDA as var_4
EBP      saved value of EBP
EBP+4    return address
EBP+8    argument#1, marked in IDA as arg_0
EBP+0xC  argument#2, marked in IDA as arg_4
EBP+0x10 argument#3, marked in IDA as arg_8
...
```

在我们的例子中，scanf()有两个参数。

第一个参数是指向"%d"的字符串指针，第二个是变量x的地址。

首先， `lea eax, DWORD PTR _x$[ebp]` 指令将变量x的地址放入EAX寄存器。LEA作用是"取有效地址"，通常用来生成一个地址（A.6.2）。

可以说，LEA在这里只是把EBP的值与宏 _x$的值相加，并存储在EAX寄存器中。

`lea eax, [ebp-4]` 的作用也是一样。

EBP的值减去4，结果放在EAX寄存器中。接着EAX寄存器的值被压入栈中，再调用printf()。

之后，printf()被调用。第一个参数是一个字符串指针："You entered %d … "。

第二个参数是通过mov ecx, [ebp-4]使用的，这个指令把变量x的内容传给ECX而不是它的地址。

然后，ECX的值放入栈中，接着最后一次调用printf()。

7.1.3 MSVC+OllyDbg

我们在OllyDbg中使用这个例子。首先载入程序，按F8直到进入我们的可执行文件而不是ntdll.dll。往下滚动屏幕找到main()。点击第一条指令（PUSH EBP），按F2设置断点，再按F9执行，触发main()开始处的断点。

让我们来跟随到准备变量x的地址的位置。

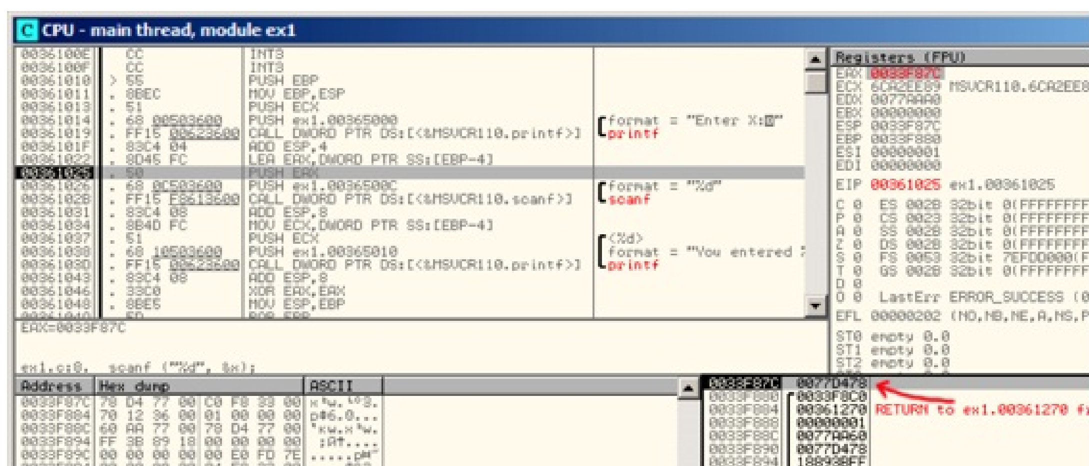


图7.1 OllyDbg：计算局部变量的地址

可以右击寄存器窗口的EAX，再点击"堆栈窗口中跟随"。这个地址会在堆栈窗口中显示。观察，这是局部栈中的一个变量。我在图中用红色箭头标出。这里是一些无用数据（0x77D478）。PUSH指令将会把这个栈元素的地址压入栈中。然后按F8直到scanf()函数执行完。在scanf()执行时，我们要在命令行窗口中输入，例如输入123。

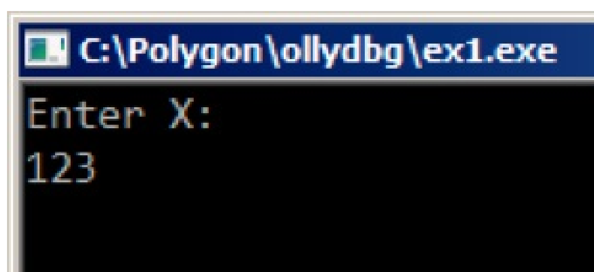


图7.2 命令行输出

scanf()在这里执行。

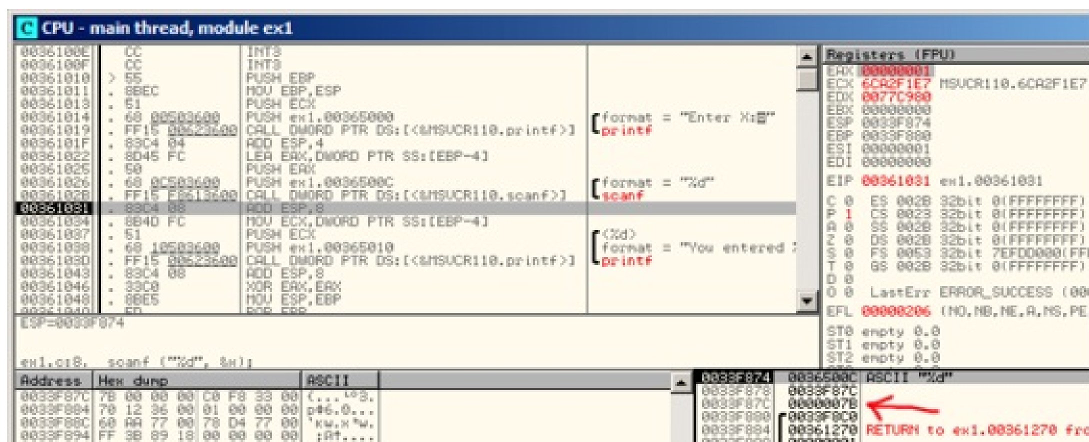


图7.3：OllyDbg：scanf()执行

scanf()在EAX中返回1，这意味着成功读入了一个值。现在我们关心的那个栈元素中的值是0x7B(123)。

接下来，这个值从栈中复制到ECX寄存器中，然后传递给printf()。

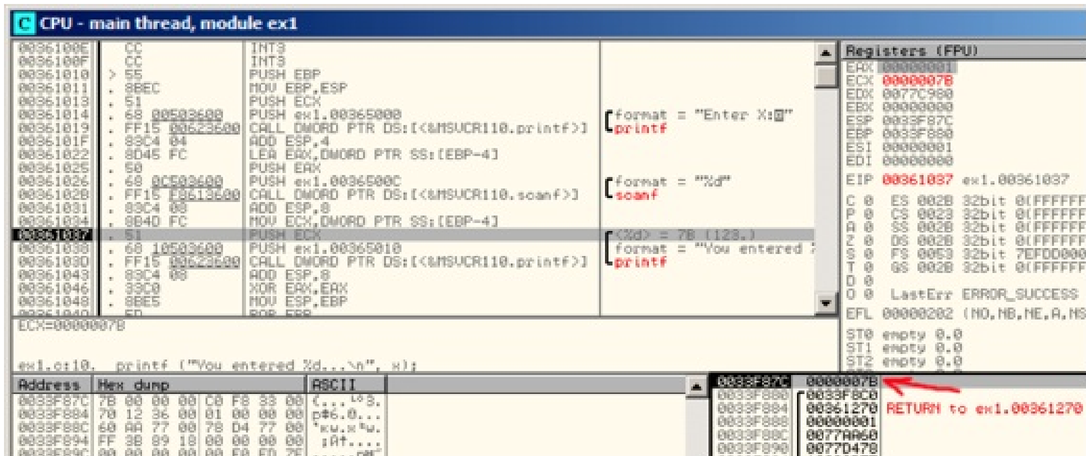


图7.4：OllyDbg：准备把值传递给printf()

GCC

让我们在Linux GCC 4.4.1下编译这段代码

```
main                proc near
var_20              = dword ptr -20h
var_1C              = dword ptr -1Ch
var_4               = dword ptr -4
                    push    ebp
                    mov     ebp, esp
                    and     esp, 0FFFFFFF0h
                    sub     esp, 20h
                    mov     [esp+20h+var_20], offset aEnterX ; "Ente
r X:"
                    call    _puts
                    mov     eax, offset aD  ; "%d"
                    lea     edx, [esp+20h+var_4]
                    mov     [esp+20h+var_1C], edx
                    mov     [esp+20h+var_20], eax
                    call    ___isoc99_scanf
                    mov     edx, [esp+20h+var_4]
                    mov     eax, offset aYouEnteredD___ ; "You enter
ed %d...\n"
                    mov     [esp+20h+var_1C], edx
                    mov     [esp+20h+var_20], eax
                    call    _printf
                    mov     eax, 0
                    leave
                    retn
main                endp
```

GCC把第一个调用的printf()替换成了puts()，原因在3.4.3节中讲过了。

和在MSVC例子中一样，参数都是用MOV指令放入栈中。

By the way

顺带一说，这个简单的例子是编译器将C/C++表达式翻译成指令列表的真实演示。C/C++表达式间没任何联系。编译器并没有神奇之处，只不过把编程语言逐行翻译成对应的机器码代码而已。

7.1.4 x64

和原来一样，只是传递参数时不使用栈而使用寄存器。

MSVC

```
_DATA   SEGMENT
$SG1289 DB 'Enter X:', 0aH, 00H
$SG1291 DB '%d', 00H
$SG1292 DB 'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3:
        sub rsp, 56
        lea rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
        call printf
        lea rdx, QWORD PTR x$[rsp]
        lea rcx, OFFSET FLAT:$SG1291 ; '%d'
        call scanf
        mov edx, DWORD PTR x$[rsp]
        lea rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
        call printf
        ; return 0
        xor eax, eax
        add rsp, 56
        ret 0
main    ENDP
_TEXT   ENDS
```

GCC

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...
"
main:
        sub     rsp, 24
        mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
        call    puts
        lea     rsi, [rsp+12]
        mov     edi, OFFSET FLAT:.LC1 ; "%d"
        xor     eax, eax
        call    __isoc99_scanf
        mov     esi, DWORD PTR [rsp+12]
        mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...
"
        xor     eax, eax
        call    printf
        ; return 0
        xor     eax, eax
        add     rsp, 24
        ret
```

7.1.5 ARM

keil优化+thumb mode

```
.text:00000042      scanf_main
.text:00000042
.text:00000042      var_8 = -8
.text:00000042
.text:00000042 08 B5            PUSH    {R3,LR}
.text:00000044 A9 A0            ADR     R0, aEnterX ; "Enter X:
"
.text:00000046 06 F0 D3 F8      BL      __2printf
.text:0000004A 69 46            MOV     R1, SP
.text:0000004C AA A0            ADR     R0, aD ; "%d"
.text:0000004E 06 F0 CD F8      BL      __0scanf
.text:00000052 00 99            LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0            ADR     R0, aYouEnteredD___ ; "Y
ou entered %d...
"
.text:00000056 06 F0 CB F8      BL      __2printf
.text:0000005A 00 20            MOVS    R0, #0
.text:0000005C 08 BD            POP     {R3,PC}
```

必须把一个指向int变量的指针传递给scanf()，这样才能通过这个指针返回一个值。Int是一个32位的值，所以我们在内存中需要4字节存储，并且正好符合32位的寄存器。局部变量x的空间分配在栈中，IDA把他命名为var_8。然而并不需要分配空间，因为栈指针(SP)指向的空间可以被立即使用。所以栈指针的值被复制到R1寄存器中，然后和格式化字符串一起送入scanf()。然后LDR指令将这个值从栈中送入R1寄存器，用以送入printf()中。

用ARM-mode和Xcode LLVM编译的代码区别不大，这里略去。

ARM64

```
.LC0:
    .string "Enter X:"
.LC1:
    .string "%d"
.LC2:
    .string "You entered %d...\n"
scanf_main:
; subtract 32 from SP, then save FP and LR in stack frame:
    stp x29, x30, [sp, -32]!
; set stack frame (FP=SP)
    add x29, sp, 0
; load pointer to the "Enter X:" string
    adrp    x0, .LC0
    add x0, x0, :lo12:.LC0
; X0=pointer to the "Enter X:" string
; print it:
    bl  puts
; load pointer to the "\%d" string:
    adrp    x0, .LC1
    add x0, x0, :lo12:.LC1
; find a space in stack frame for "x" variable (X1=FP+28):
    add x1, x29, 28
; X1=address of "x" variable'
; pass the address to scanf() and call it:
    bl  __isoc99_scanf
; load 32-bit value from the variable in stack frame:
    ldr w1, [x29,28]
; W1=x
; load pointer to the "You entered \%d...\n" string
; printf() will take text string from X0 and "x" variable from X
1 (or W1)
    adrp    x0, .LC2
    add x0, x0, :lo12:.LC2
    bl  printf
; _return 0
    mov w0, 0
; restore FP and LR, then add 32 to SP:
    ldp x29, x30, [sp], 32
    ret
```

在栈帧上申请了32字节空间，比它需要的要大，可能是因为内存地址对齐问题？最有趣的是寻找栈帧上x变量的空间(代码22行)，为什么是加28?因为编译器是在栈帧的结束而不是开始的时间决定变量的空间。传递给scanf()的地址上储存这用户输入的值。32位值的类型是int，在代码27行中拿到然后传递给printf()。

7.1.6 MIPS

用$sp+24指向栈上申请的x变量的地址，然后将地址传给scanf()，用户输入的值使用LW(Load Word)指令传递给printf()。

Listing 7.4: Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
$LC1:
$LC2:
.ascii  "Enter X:\000"
.ascii  "%d\000"
.ascii  "You entered %d...\012\000"
main:
; function prologue:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-40
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,36($sp)
; call puts():
        lw      $25,%call16(puts)($28)
        lui     $4,%hi($LC0)
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; branch delay slot
; call scanf():
        lw      $28,16($sp)
        lui     $4,%hi($LC1)
        lw      $25,%call16(__isoc99_scanf)($28)
; set 2nd argument of scanf(), $a1=$sp+24:
        addiu   $5,$sp,24
        jalr    $25
        addiu   $4,$4,%lo($LC1) ; branch delay slot
; call printf():
        lw      $28,16($sp)
; set 2nd argument of printf(),
; load word at address $sp+24:
```

IDA 中显示的栈帧情况如下：

```
.text:00000000 main:
.text:00000000
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
```

```
.text:00000000
; function prologue:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 1
6)
.text:00000004                 addiu   $sp, -0x28
.text:00000008                 la      $gp, (__gnu_local_gp & 0x
FFFF)
.text:0000000C                 sw      $ra, 0x28+var_4($sp)
.text:00000010                 sw      $gp, 0x28+var_18($sp)
; call puts():
.text:00000014                 lw      $t9, (puts & 0xFFFF)($gp)
.text:00000018                 lui     $a0, ($LC0 >> 16)  # "Ent
er X:"
.text:0000001C                 jalr    $t9
.text:00000020                 la      $a0, ($LC0 & 0xFFFF)  # "
Enter X:" ; branch delay slot
; call scanf():
.text:00000024                 lw      $gp, 0x28+var_18($sp)
.text:00000028                 lui     $a0, ($LC1 >> 16)  # "%d"
.text:0000002C                 lw      $t9, (__isoc99_scanf & 0x
FFFF)($gp)
; set 2nd argument of scanf(), $a1=$sp+24:
.text:00000030                 addiu   $a1, $sp, 0x28+var_10
.text:00000034                 jalr    $t9  ; branch delay slot
.text:00000038                 la      $a0, ($LC1 & 0xFFFF)  # "
%d"
; call printf():
.text:0000003C                 lw      $gp, 0x28+var_18($sp)
; set 2nd argument of printf(),
; load word at address $sp+24:
.text:00000040                 lw      $a1, 0x28+var_10($sp)
.text:00000044                 lw      $t9, (printf & 0xFFFF)($g
p)
.text:00000048                 lui     $a0, ($LC2 >> 16)  # "You
 entered %d...\n"
.text:0000004C                 jalr    $t9
.text:00000050                 la      $a0, ($LC2 & 0xFFFF)  # "
You entered %d...\n" ; branch delay slot
; function epilogue:
.text:00000054                 lw      $ra, 0x28+var_4($sp)
; set return value to 0:
.text:00000058                 move    $v0, $zero
; _return:
.text:0000005C                 jr      $ra
.text:00000060                 addiu   $sp, 0x28 ; branch delay
slot
```

7.2 全局变量

如果之前的例子中的x变量不再是本地变量而是全局变量呢？那么就有可能从任何地方访问它，不仅仅是函数体，全局变量被认为anti-pattern(通常被认为是一个不好的习惯)，但是为了试验，我们可以这样做。

```
#include <stdio.h>
int x;
int main()
{
    printf ("Enter X:
");
    scanf ("%d", &x);
    printf ("You entered %d...
", x);
    return 0;
};
```

7.2.1 MSVC: x86

```
_DATA       SEGMENT
COMM        _x:DWORD
$SG2456     DB      'Enter X:', 0aH, 00H
$SG2457     DB      '%d', 00H
$SG2458     DB      'You entered %d...', 0aH, 00H
_DATA   ENDS
PUBLIC  _main
EXTRN   _scanf:PROC
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2456
    call    _printf
    add     esp, 4
    push    OFFSET _x
    push    OFFSET $SG2457
    call    _scanf
    add     esp, 8
    mov     eax, DWORD PTR _x
    push    eax
    push    OFFSET $SG2458
    call    _printf
    add     esp, 8
    xor     eax, eax
    pop     ebp
    ret     0
_main ENDP
_TEXT ENDS
```

现在x变量被定义为在_DATA部分，局部堆栈不允许再分配任何内存，除了直接访问内存所有通过栈的访问都不被允许。在执行的文件中全局变量还未初始化(实际上，我们为什么要在执行文件中为未初始化的变量分配一块？)但是当访问这里时，系统会在这里分配一块0值。

现在让我们来分析变量的分配。

```
int x=10; // default value
```

我们得到:

```
_DATA   SEGMENT
_x      DD      0aH
...
```

这里我们看见一个双字节的值0xA(DD 表示双字节 = 32bit)

如果你在IDA中打开compiled.exe，你会发现x变量被放置在_DATA块的开始处，接着你就会看见文本字符串。

如果你在IDA中打开之前例子中的compiled.exe中X变量没有定义的地方，你就会看见像这样的东西:

```
.data:0040FA80 _x                dd ?         ; DATA XREF: _main+1
0
.data:0040FA80                                ; _main+22
.data:0040FA84 dword_40FA84      dd ?         ; DATA XREF: _memset
+1E
.data:0040FA84                                ; unknown_libname_1+
28
.data:0040FA88 dword_40FA88      dd ?         ; DATA XREF: ___sbh_
find_block+5
.data:0040FA88                                ; ___sbh_free_block+
2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem             dd ?         ; DATA XREF: ___sbh_
find_block+B
.data:0040FA8C                                ; ___sbh_free_block+
2CA
.data:0040FA90 dword_40FA90      dd ?         ; DATA XREF: _V6_Hea
pAlloc+13
.data:0040FA90                                ; __calloc_impl+72
.data:0040FA94 dword_40FA94      dd ?         ; DATA XREF: ___sbh_
free_block+2FE
```

被_x替换了？其它变量也并未要求初始化，这也就是说在载入exe至内存后，在这里有一块针对所有变量的空间，并且还有一些随机的垃圾数据。但在exe中这些没有初始化的变量并不影响什么，比如它适合大数组。

7.2.2 MSVC: x86 + OllyDbg

到这里事情就变得简单了

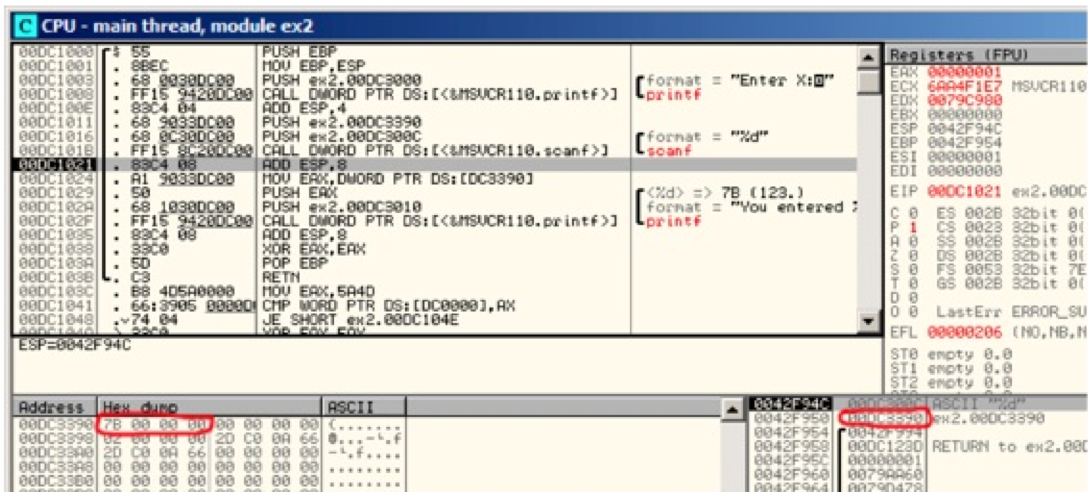


表7.5 OllyDbg: scanf()执行后

变量都在data部分里，在PUSH指令(压入x的地址)被执行后，地址将会在栈中显示，那么右击元组数据，点击"FIlow in dump"，然后变量就会在左侧内存窗口显示.

在命令行窗口中输入123后，这里就会显示0x7B

但是为什么第一个字节是7B?合理的猜测，这里会有一组00 00 7B，被称为是字节顺序，然后在x86中使用的是小端，也就是说低位字节先写，高位字节后写。

回到例子中，这里的32-bit值就会载入到EAX中，然后被传递给printf().

X变量地址是0xDC3390。在OllyDbg中我们看进程内存映射(Alt-M)，然后发现这个地址在PE文件.data结构处。

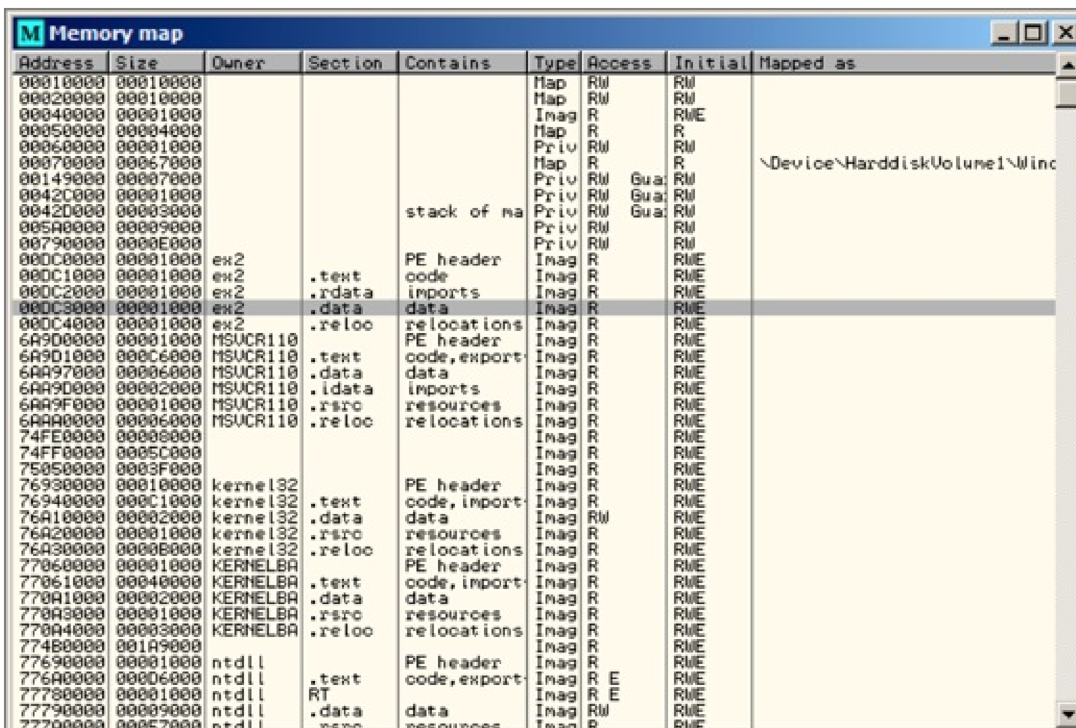


表7.6: OllyDbg 进程内存映射

7.2.3 GCC: x86

这和linux中几乎是一样的，除了segment的名称和属性:未初始化变量被放置在_bss部分。

在ELF文件格式中，这部分数据有这样的属性:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

如果静态的分配一个值，比如10，它将会被放在_data部分，这部分有下面的属性:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

7.2.4 MSVC: x64

```
_DATA       SEGMENT
COMM        x:DWORD
$SG2924     DB      'Enter X:', 0aH, 00H
$SG2925     DB      '%d', 00H
$SG2926     DB      'You entered %d...', 0aH, 00H
_DATA       ENDS

_TEXT       SEGMENT
main        PROC
$LN3:
            sub     rsp, 40
            lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
            call    printf
            lea     rdx, OFFSET FLAT:x
            lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
            call    scanf
            mov     edx, DWORD PTR x
            lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d..
.'
            call    printf
            ; return 0
            xor     eax, eax
            add     rsp, 40
            ret     0
main ENDP
_TEXT ENDS
```

几乎和x86中的代码是一样的，注意x变量的地址传递给scanf()用的是LEA指令，尽管第二处传递给printf()变量时用的是MOV指令，"DWORD PTR"——是汇编语言中的一部分(和机器码没有联系)。这就表示变量数据类型是32-bit，于是MOV指令就被编码了。

7.2.5 ARM:Optimizing Keil 6/2013 (Thumb mode)

```
.text:00000000 ; Segment type: Pure code
.text:00000000          AREA .text, CODE
...
.text:00000000 main
.text:00000000                  PUSH    {R4,LR}
.text:00000002                  ADR     R0, aEnterX         ; "E
nter X:
"
.text:00000004                  BL      __2printf
.text:00000008                  LDR     R1, =x
.text:0000000A                  ADR     R0, aD              ; "%
d"
.text:0000000C                  BL      __0scanf
.text:00000010                  LDR     R0, =x
.text:00000012                  LDR     R1, [R0]
.text:00000014                  ADR     R0, aYouEnteredD___ ; "Y
ou entered %d...
"
.text:00000016                  BL      __2printf
.text:0000001A                  MOVS    R0, #0
.text:0000001C                  POP     {R4,PC}
...
.text:00000020 aEnterX          DCB     "Enter X:",0xA,0    ; DA
TA XREF: main+2
.text:0000002A                  DCB     0
.text:0000002B                  DCB     0
.text:0000002C off_2C           DCD x                       ; DA
TA XREF: main+8
.text:0000002C                                      ; main+10
.text:00000030 aD               DCB     "%d",0              ; DA
TA XREF: main+A
.text:00000033                  DCB 0
.text:00000034 aYouEnteredD___  DCB "You entered %d...",0xA,0 ;
DATA XREF: main+14
.text:00000047                  DCB 0
.text:00000047 ; .text          ends
.text:00000047
...
.data:00000048 ; Segment type:  Pure data
.data:00000048                  AREA .data, DATA
.data:00000048                  ; ORG 0x48
.data:00000048                  EXPORT x
.data:00000048 x                DCD 0xA                     ; DA
TA XREF: main+8
.data:00000048                                              ; ma
in+10
.data:00000048                                              ; .d
ata ends
```

那么，现在x变量以某种方式变为全局的，现在被放置在另一个部分中。命名为data块(.data)。有人可能会问，为什么文本字符串被放在了代码块(.text)，而且x可以被放在这？因为这是变量，而且根据它的定义，它可以变化，也有可能会频繁变化，不频繁变化的代码块可以被放置在ROM中，变化的变量在RAM中，当有ROM时在RAM中储存不变的变量是不利于节约资源的。

此外，RAM中数据部分常量必须在之前初始化，因为在RAM使用后，很明显，将会包含杂乱的信息。

继续向前，我们可以看到，在代码片段，有个指针指向X变量(0ff_2C)。然后所有关于变量的操作都是通过这个指针。这也是x变量可以被放在远离这里地方的原因。所以他的地址一定被存在离这很近的地方。LDR指令在thumb模式下只可访问指向地址在1020bytes内的数据。同样的指令在ARM模式下——范围就达到了-4095bytes~ +4095bytes，也就是x变量地址一定要在这附近的原因。因为没法保证链接时会把这个变量放在附近，它甚至可能在外部存储芯片里！

另外，如果变量以const声明，Keil编译环境下则会将变量放在.constdata部分，大概从那以后，链接时就可以把这部分和代码块放在ROM里了。

7.2.6 ARM64

```
    .comm   x,4,4
.LC0:
    .string "Enter X:"
.LC1:
    .string "%d"
.LC2:
    .string "You entered %d...\n"
f5:
; save FP and LR in stack frame:
    stp x29, x30, [sp, -16]!
; set stack frame (FP=SP)
    add x29, sp, 0
; load pointer to the "Enter X:" string:
    adrp    x0, .LC0
    add x0, x0, :lo12:.LC0
    bl  puts
; load pointer to the "\%d" string:
    adrp    x0, .LC1
    add x0, x0, :lo12:.LC1
; form address of x global variable:
    adrp    x1, x
    add x1, x1, :lo12:x
    bl  __isoc99_scanf
; form address of x global variable again:
    adrp    x0, x
    add x0, x0, :lo12:x
; load value from memory at this address:
    ldr w1, [x0]
; load pointer to the "You entered \%d...\n" string:
    adrp    x0, .LC2
    add x0, x0, :lo12:.LC2
    bl  printf
; _return 0
    mov w0, 0
; restore FP and LR:
    ldp x29, x30, [sp], 16
    ret
```

本例中，x变量声明为了全局变量，它的地址通过 ADRP/ADD 指令对来计算。(代码 21 和 25 行).

7.2.7 MIPS

未初始化的全局变量

已初始化的全局变量

7.3 scanf()结果检查

正如我之前所见的，现在使用scanf()有点过时了，但是如过我们不得不这样做时，我们需要检查scanf()执行完毕时是否发生了错误。

```
#include <stdio.h>
int main()
{
    int x;
    printf ("Enter X:
");

    if (scanf ("%d", &x)==1)
        printf ("You entered %d...
", x);
    else
        printf ("What you entered? Huh?
");

    return 0;
};
```

按标准，scanf()函数返回成功获取的字段数。

在我们的例子中，如果事情顺利，用户输入一个数字，scanf()将会返回1或0或者错误情况下返回EOF.

这里，我们添加了一些检查scanf()结果的c代码，用来打印错误信息:

按照预期的回显:

```
C:...>ex3.exe
Enter X:
123
You entered 123...

C:...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

7.3.1 MSVC: x86

我们可以得到这样的汇编代码(msvc2010):

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH,
00H
        call    _printf
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

调用函数(main())必须能够访问到被调用函数(scanf())的结果，所以callee把这个值留在了EAX寄存器中。

然后我们在"CMP EAX, 1"指令的帮助下，换句话说，我们将eax中的值与1进行比较。

JNE根据CMP的结果判断跳至哪，JNE表示(jump if Not Equal)

所以，如果EAX中的值不等于1，那么处理器就会将执行流程跳转到JNE指向的，在我们的例子中是$LN2@main，当流程跳到这里时，CPU将会带着参数"What you entered? Huh?"执行printf(),但是执行正常，就不会发生跳转，然后另外一个printf()就会执行，两个参数为"You entered %d…"及x变量的值。

因为第二个printf()并没有被执行，后面有一个JMP(无条件跳转)，就会将执行流程到第二个printf()后"XOR EAX, EAX"前，执行完返回0。

那么，可以这么说，比较两个值通常使用CMP/Jcc这对指令，cc是条件码，CMP比较两个值，然后设置processor flag，Jcc检查flags然后判断是否跳。

但是事实上，这却被认为是诡异的。但是CMP指令事实上,但是CMP指令实际上是SUB(subtract),所有算术指令都会设置processor flags,不仅仅只有CMP，当我们比较1和1时，1结果就变成了0，ZF flag就会被设定(表示最后一次的比较结果为0)，除了两个数相等以外，再没有其他情况了。JNE 检查ZF flag，如果没有设定就会跳转。JNE实际上就是JNZ(Jump if Not Zero)指令。JNE和JNZ的机器码都是一样的。所以CMP指令可以被SUB指令代替，几乎一切的都没什么变化。但是SUB会改变第一个数，CMP是"SUB without saving result".

7.3.2 MSVC: x86:IDA

现在是时候打开IDA然后尝试做些什么了，顺便说一句。对于初学者来说使用在MSVC中使用/MD是个非常好的主意。这样所有独立的函数不会从可执行文件中link，而是从MSVCR*.dll。因此这样可以简单明了的发现函数在哪里被调用。

当在IDA中分析代码时，建议一定要做笔记。比如在分析这个例子的时候，我们看到了JNZ将要被设置为error，所以点击标注，然后标注为"error"。另外一处标注在"exit":

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4          = dword ptr -4
.text:00401000 argc           = dword ptr 8
.text:00401000 argv           = dword ptr 0Ch
.text:00401000 envp           = dword ptr 10h
.text:00401000
.text:00401000                push    ebp
.text:00401001                mov     ebp, esp
.text:00401003                push    ecx
.text:00401004                push    offset Format   ; "Enter X:
"
.text:00401009                call    ds:printf
.text:0040100F                add     esp, 4
.text:00401012                lea     eax, [ebp+var_4]
.text:00401015                push    eax
.text:00401016                push    offset aD       ; "%d"
.text:0040101B                call    ds:scanf
.text:00401021                add     esp, 8
.text:00401024                cmp     eax, 1
.text:00401027                jnz     short error
.text:00401029                mov     ecx, [ebp+var_4]
.text:0040102C                push    ecx
.text:0040102D                push    offset aYou     ; "You enter
ed %d...
"
.text:00401032                call    ds:printf
.text:00401038                add     esp, 8
.text:0040103B                jmp     short exit
.text:0040103D ; -----------------------------------------------
----------------------------
.text:0040103D
.text:0040103D error:                                 ; CODE XREF:
 _main+27
.text:0040103D                push    offset aWhat    ; "What you
entered? Huh?
"
.text:00401042                call    ds:printf
.text:00401048                add     esp, 4
.text:0040104B
.text:0040104B exit:                                  ; CODE XREF:
 _main+3B
.text:0040104B                xor     eax, eax
.text:0040104D                mov     esp, ebp
.text:0040104F                pop     ebp
.text:00401050                retn
.text:00401050 _main   endp
```

现在理解代码就变得非常简单了。然而过分的标注指令却不是一个好主意。

函数的一部分有可能也会被IDA隐藏:

我隐藏了两部分然后分别给它们命名:

```
.text:00401000 _text           segment para public ’CODE’ use32
.text:00401000                 assume cs:_text
.text:00401000                 ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024                 cmp     eax, 1
.text:00401027                 jnz     short error
.text:00401029 ; print result
.text:0040103B                 jmp     short exit
.text:0040103D ; -----------------------------------------------
----------------------------
.text:0040103D
.text:0040103D error:                                   ; CODE XREF:
 _main+27
.text:0040103D                 push    offset aWhat    ; "What you
entered? Huh?
"
.text:00401042                 call    ds:printf
.text:00401048                 add     esp, 4
.text:0040104B
.text:0040104B exit:                                    ; CODE XREF:
 _main+3B
.text:0040104B                 xor     eax, eax
.text:0040104D                 mov     esp, ebp
.text:0040104F                 pop     ebp
.text:00401050                 retn
.text:00401050 _main           endp
```

如果要显示这些隐藏的部分，我们可以点击数字上的+。

为了压缩"空间"，我们可以看到IDA怎样用图表代替一个函数的(见图6.7)，然后在每个条件跳转处有两个箭头，绿色和红色。绿色箭头代表如果跳转触发的方向，红色则相反。

当然可以折叠节点，然后备注名称,我像这样处理了3块(见图 6.8):

这个非常的有用。可以这么说，逆向工程师很重要的一点就是缩小他所有的信息。

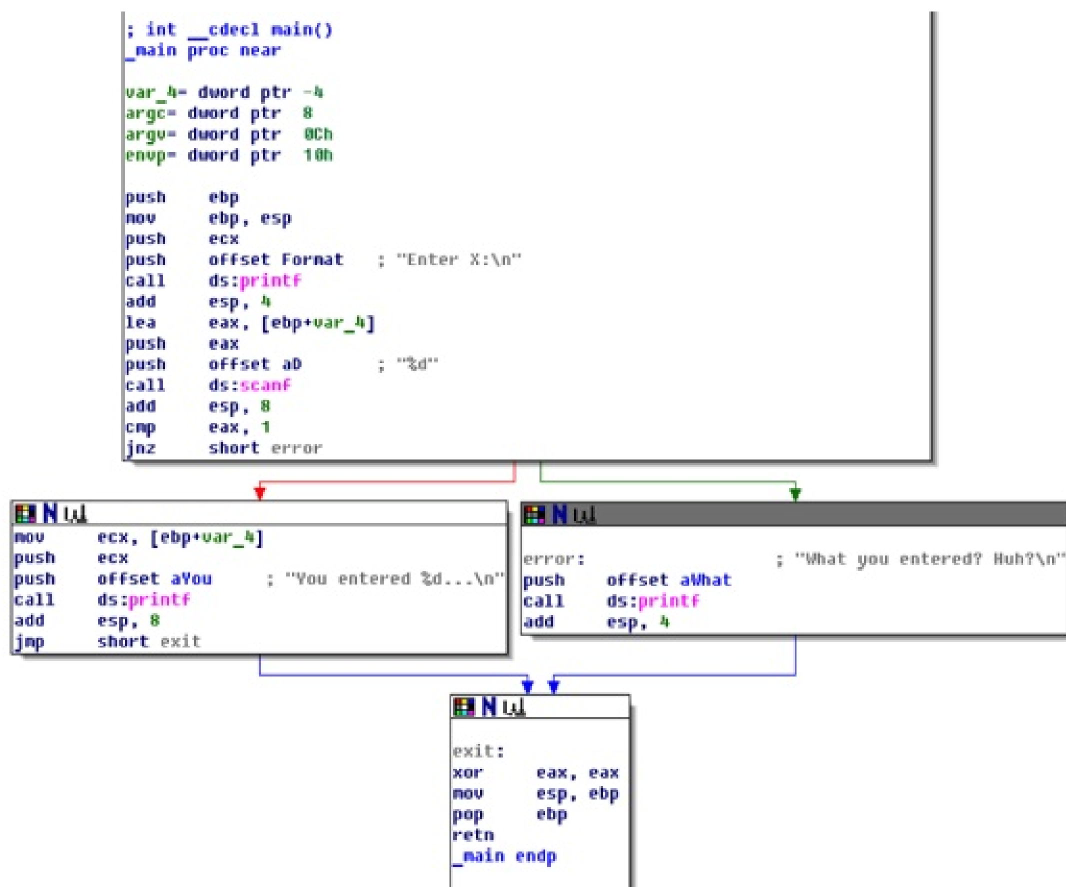


图7.7: IDA 图形模式

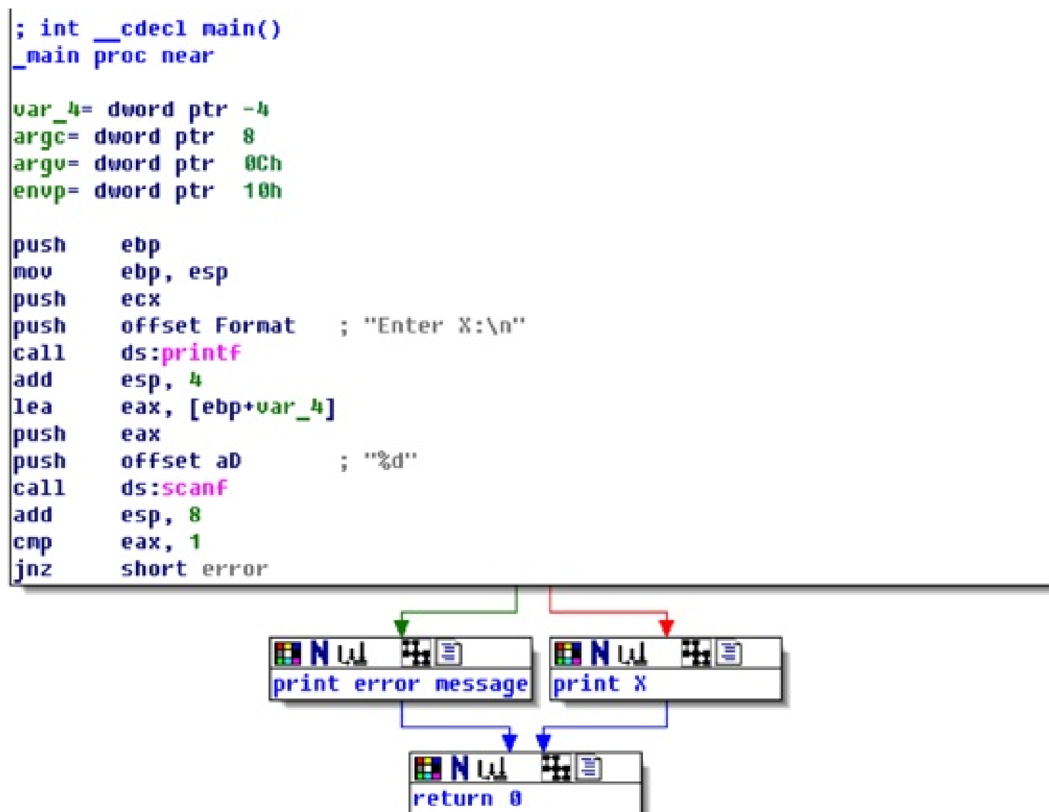


图7.8: Graph mode in IDA with 3 nodes folded

7.3.3 MSVC: x86 + OllyDbg

让我们继续在OllyDbg中看这个范例程序，使它认为scanf()怎么运行都不会出错。

当本地变量地址被传递给scanf()时，这个变量还有一些垃圾数据。这里是0x4CD478:

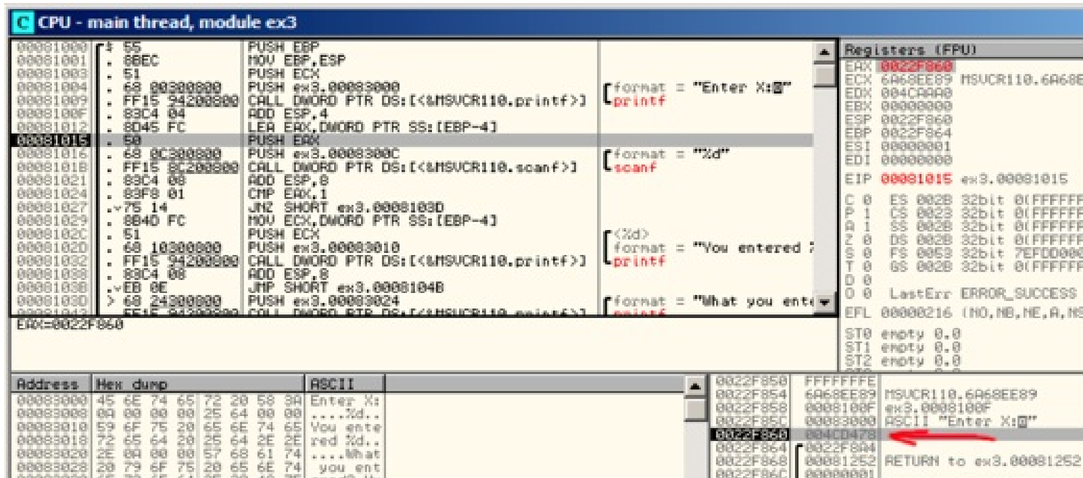


图7.10 OllyDbg：传递变量地址给printf()

当scanf()执行时，我在命令行窗口输入了一些不是数字的东西，像"asdasd".scanf()结束后eax变为了0.也就意味着有错误发生:

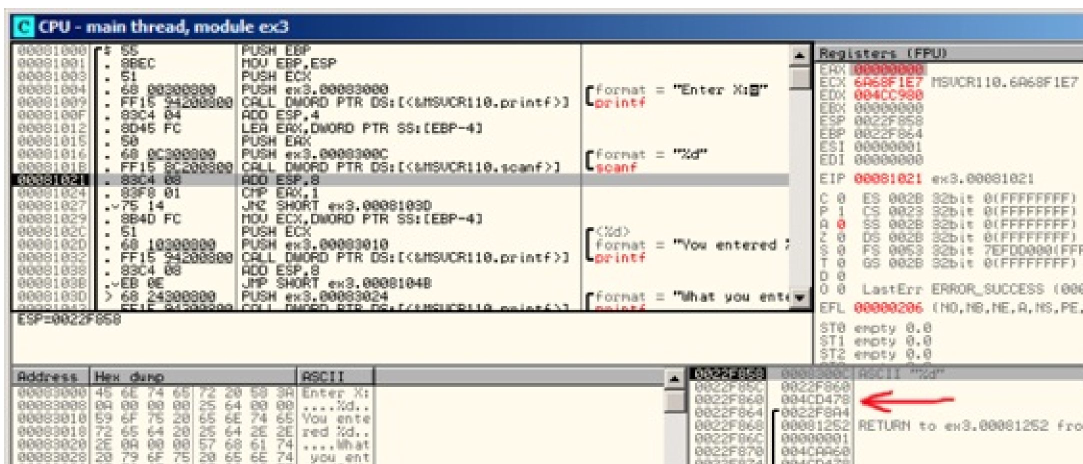


图7.11 OllyDbg：scanf() 返回错误

我们也可以发现栈中的本地变量并没有发生变化，scanf()会在那里写入什么呢？其实什么都没有，只是返回了0.

现在让我们尝试修改这个程序，右击EAX，在选项中有个"set to 1"，这正是我们所需要的。

现在EAX是1了。那么接下来的检查就会按照我们的需求执行，然后printf()将会打印出栈上的变量。

按下F9我们可以在窗口中看到:

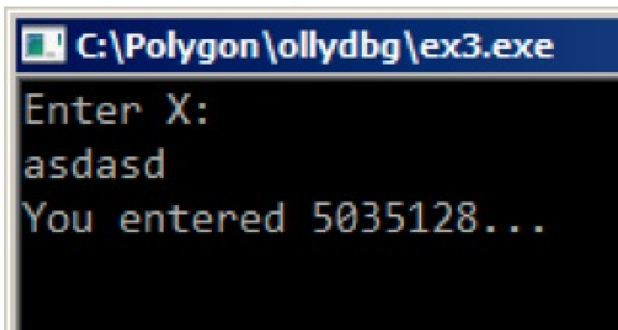


图7.9

实际上，5035128是栈上一个数据(0x4CD478)的十进制表示!

7.3.4 MSVC: x86 + HIew

这也是一个关于可执行文件patch的简单例子，我们之前尝试patch程序，所以程序总是打印数字，不管我们输入什么。

假设编译时并没有使用/MD,我们可以在.text开始的地方找到main()函数，现在让我们在Hiew中打开执行文件。找到.text的开始处(enter,F8,F6,enter,enter)

我们可以看到这个:

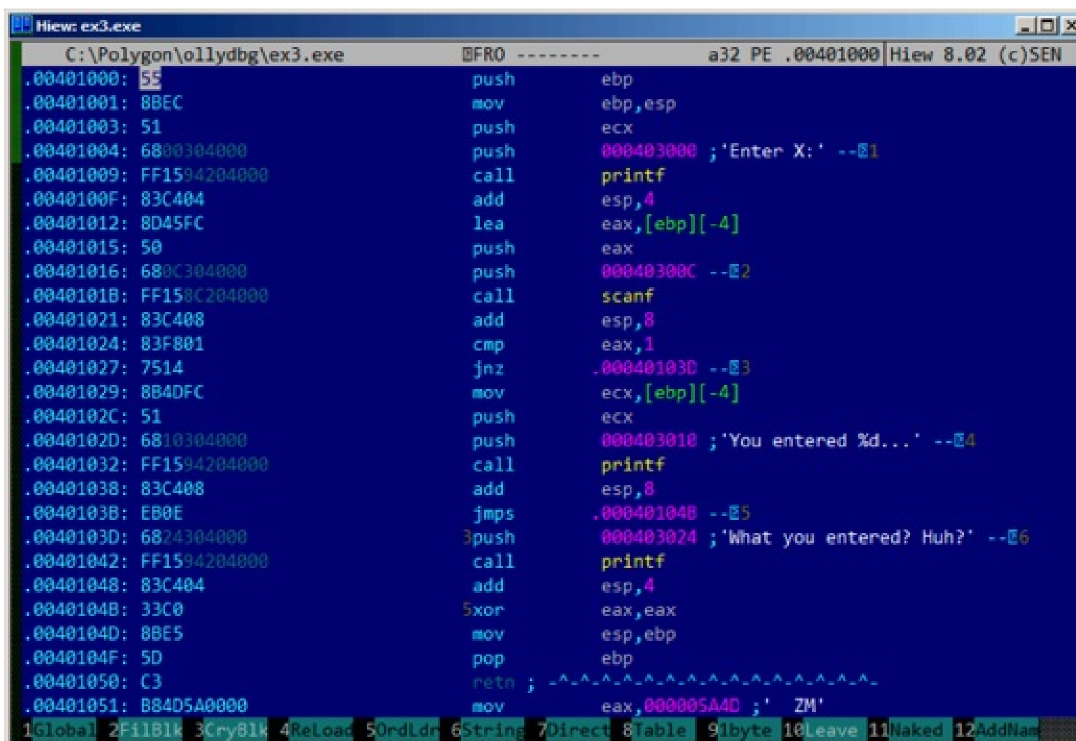


图7.12:main()函数

Hiew 找到 ASCIIZ 字符串并显示，引入的函数名字也同样显示。

移动光标到地址 .00401027 (这里是 JNZ 指令, 我们需要绕过它), 按下 F3, 然后输入 “9090” (表示两个 NOP):

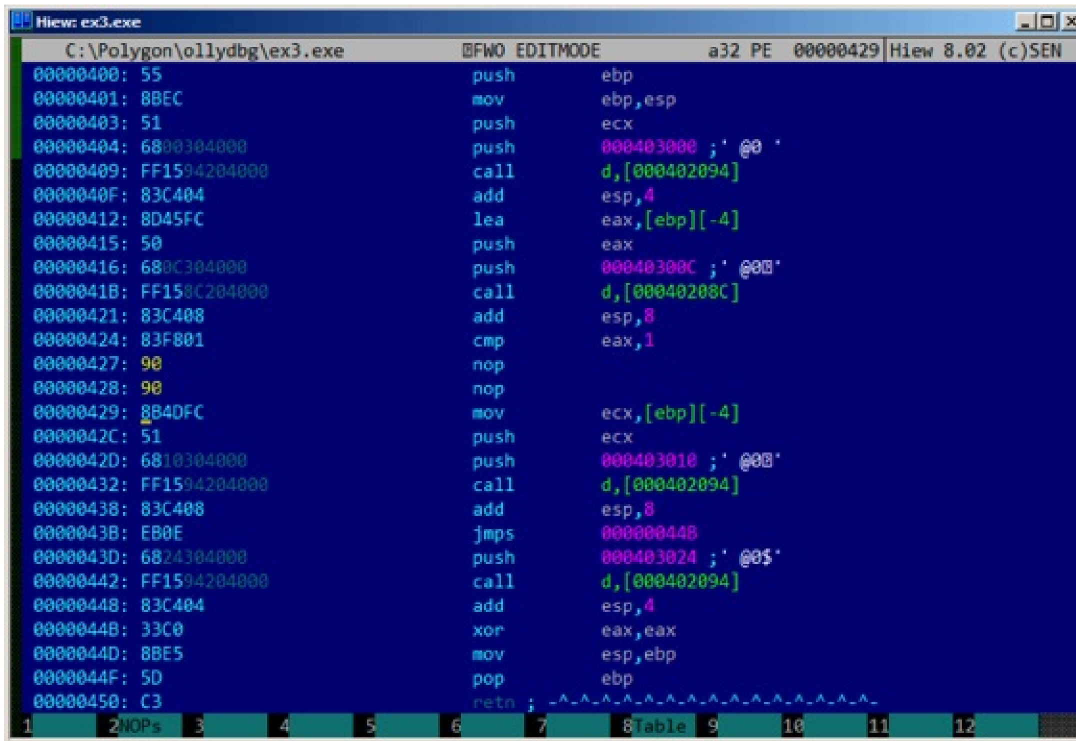


图7.13:Hiew 用两个NOP替换JNZ

然后按下F9(update),现在文件保存在了磁盘中，它将会按照我们希望的那样执行。

两个NOP可能看起来并不是那么完美，另一个方法是把0写在第二处（jump offset）,所以JNZ就可以总是跳到下一个指令了。

另外我们也可以这样做：替换第一个字节为EB，这样就不修改第二处（jump offset），这样就会无条件跳转，不管我们输入什么，错误信息都可以打印出来了。

7.3.5 MSVC: x64

因为我们这里处理的是无整型变量。在x86-64中还是32bit,我们可以看出32bit的寄存器(前缀为E)在这种情况下是怎样使用的,然而64bit的寄存也有被使用(前缀R)

```
_DATA       SEGMENT
$SG2924     DB      'Enter X:', 0aH, 00H
$SG2926     DB      '%d', 00H
$SG2927     DB      'You entered %d...', 0aH, 00H
$SG2929     DB      'What you entered? Huh?', 0aH, 00H
_DATA       ENDS

_TEXT       SEGMENT
x$ = 32
main        PROC
$LN5:
            sub         rsp, 56
            lea         rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
            call        printf
            lea         rdx, QWORD PTR x$[rsp]
            lea         rcx, OFFSET FLAT:$SG2926 ; '%d'
            call        scanf
            cmp         eax, 1
            jne         SHORT $LN2@main
            mov         edx, DWORD PTR x$[rsp]
            lea         rcx, OFFSET FLAT:$SG2927 ; 'You entered
%d...'
            call        printf
            jmp         SHORT $LN1@main
$LN2@main:
            lea rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Hu
h?'
            call printf
$LN1@main:
            ; return 0
            xor         eax, eax
            add         rsp, 56
            ret         0
main        ENDP
_TEXT       ENDS
END
```

7.3.6 ARM

ARM:Optimizing Keil 6/2013 (Thumb mode)

```
var_8       = -8

            PUSH    {R3,LR}
            ADR     R0, aEnterX         ; "Enter X:
"
            BL      __2printf
            MOV     R1, SP
            ADR     R0, aD              ; "%d"
            BL      __0scanf
            CMP     R0, #1
            BEQ     loc_1E
            ADR     R0, aWhatYouEntered ; "What you entered? Huh
?
"
            BL      __2printf
loc_1A                                  ; CODE XREF: main+26
            MOVS    R0, #0
            POP     {R3,PC}

loc_1E                                  ; CODE XREF: main+12
            LDR     R1, [SP,#8+var_8]
            ADR     R0, aYouEnteredD___ ; "You entered %d...
"
            BL      __2printf
            B       loc_1A
```

这里有两个新指令CMP 和BEQ.

CMP和x86指令中的相似，它会用一个参数减去另外一个参数然后保存flag.

BEQ是跳向另一处地址，如果数相等就会跳，如果最后一次比较结果为0，或者Z flag是1。和x86中的JZ是一样的。

其他的都很简单，执行流程分为两个方向，当R0被写入0后，两个方向则会合并，作为函数的返回值，然后函数结束。

ARM64

```
.LC0:
    .string "Enter X:"
.LC1:
    .string "%d"
.LC2:
    .string "You entered %d...\n"
.LC3:
    .string "What you entered? Huh?"
f6:
; save FP and LR in stack frame:
    stp x29, x30, [sp, -32]!
; set stack frame (FP=SP)
    add x29, sp, 0
; load pointer to the "Enter X:" string:
    adrp    x0, .LC0
    add x0, x0, :lo12:.LC0
    bl  puts
; load pointer to the "\%d" string':
    adrp    x0, .LC1
    add x0, x0, :lo12:.LC1
; calculate address of x variable in the local stack
    add x1, x29, 28
    bl  __isoc99_scanf
; scanf() returned result in W0.
; check it:
    cmp w0, 1
; BNE _is Branch if Not Equal
; so if W0<>0, jump to L2 will be occurred
    bne .L2
; at this moment W0=1, meaning no error
; load x value from the local stack
    ldr w1, [x29,28]
; load pointer to the "You entered \%d...\n" string:
    adrp    x0, .LC2
    add x0, x0, :lo12:.LC2
    bl  printf
; skip the code, which print the "What you entered? Huh?" string
:
    b   .L3
.L2:
; load pointer to the "What you entered? Huh?" string:
    adrp    x0, .LC3
    add x0, x0, :lo12:.LC3
    bl  puts
.L3:
; _return 0
    mov w0, 0
; restore FP and LR:
    ldp x29, x30, [sp], 32
    ret
```

7.3.7 MIPS

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18          = -0x18
.text:004006A0 var_10          = -0x10
.text:004006A0 var_4           = -4
.text:004006A0
.text:004006A0                 lui     $gp, 0x42
.text:004006A4                 addiu   $sp, -0x28
.text:004006A8                 li      $gp, 0x418960
.text:004006AC                 sw      $ra, 0x28+var_4($sp)
.text:004006B0                 sw      $gp, 0x28+var_18($sp)
.text:004006B4                 la      $t9, puts
.text:004006B8                 lui     $a0, 0x40
.text:004006BC                 jalr    $t9 ; puts
.text:004006C0                 la      $a0, aEnterX     # "Enter
 X:"
.text:004006C4                 lw      $gp, 0x28+var_18($sp)
.text:004006C8                 lui     $a0, 0x40
.text:004006CC                 la      $t9, __isoc99_scanf
.text:004006D0                 la      $a0, aD          # "%d"
.text:004006D4                 jalr    $t9 ; __isoc99_scanf
.text:004006D8                 addiu   $a1, $sp, 0x28+var_10  #
branch delay slot
.text:004006DC                 li      $v1, 1
.text:004006E0                 lw      $gp, 0x28+var_18($sp)
.text:004006E4                 beq     $v0, $v1, loc_40070C
.text:004006E8                 or      $at, $zero       # branch
 delay slot, NOP
.text:004006EC                 la      $t9, puts
.text:004006F0                 lui     $a0, 0x40
.text:004006F4                 jalr    $t9 ; puts
.text:004006F8                 la      $a0, aWhatYouEntered  # "
What you entered? Huh?"
.text:004006FC                 lw      $ra, 0x28+var_4($sp)
.text:00400700                 move    $v0, $zero
.text:00400704                 jr      $ra
.text:00400708                 addiu   $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C                 la      $t9, printf
.text:00400710                 lw      $a1, 0x28+var_10($sp)
.text:00400714                 lui     $a0, 0x40
.text:00400718                 jalr    $t9 ; printf
.text:0040071C                 la      $a0, aYouEnteredD___  # "
You entered %d...\n"
.text:00400720                 lw      $ra, 0x28+var_4($sp)
.text:00400724                 move    $v0, $zero
.text:00400728                 jr      $ra
.text:0040072C                 addiu   $sp, 0x28
```

scanf()在$V0寄存器中返回结果。通过对比$V0和$V1的值检查地址0x004006E4。BEQ 表示 “Branch Equal”。 如果值相等 (i.e., success), 程序执行将跳至0x0040070C。

7.3.8 练习

我们可以看见,JNE/JNZ 指令可以很容易的被 JE/JZ 指令替代，反之亦然。 但是之后基础区块也被交换了。尝试在练习中做做吧。

7.4 练习

- http://challenges.re/53

第八章

访问传递参数

现在我们来看函数调用者通过栈把参数传递到被调用函数。被调用函数是如何访问这些参数呢？

```
#include <stdio.h>
int f (int a, int b, int c)
{
        return a*b+c;
};
int main()
{
        printf ("%d
", f(1, 2, 3));
        return 0;
};
```

8.1 X86

8.1.1 MSVC

如下为相应的反汇编代码（MSVC 2010 Express）

Listing 8.2 MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8                                  ; size = 4
_b$ = 12                                 ; size = 4
_c$ = 16                                 ; size = 4
_f      PROC
        push ebp
        mov ebp, esp
        mov eax, DWORD PTR _a$[ebp]
        imul eax, DWORD PTR _b$[ebp]
        add eax, DWORD PTR _c$[ebp]
        pop ebp
        ret 0
_f      ENDP

_main   PROC
        push ebp
        mov ebp, esp
        push 3 ; 3rd argument
        push 2 ; 2nd argument
        push 1 ; 1st argument
        call _f
        add esp, 12
        push eax
        push OFFSET $SG2463 ; ’%d’, 0aH, 00H
        call _printf
        add esp, 8
        ; return 0
        xor eax, eax
        pop ebp
        ret 0
_main ENDP
```

我们可以看到函数main()中3个数字被压栈，然后函数f(int, int, int)被调用。函数f()内部访问参数时使用了像_ a$=8 的宏，同样，在函数内部访问局部变量也使用了类似的形式，不同的是访问参数时偏移值（为正值）。因此EBP寄存器的值加上宏_a$的值指向压栈参数。

_a$[ebp]的值被存储在寄存器eax中，IMUL指令执行后，eax的值为eax与_b$[ebp]的乘积，然后eax与_c$[ebp]的值相加并将和放入eax寄存器中,之后返回eax的值。返回值作为printf()的参数。

8.1.2 MSVC+OllyDbg

我们在OllyDbg中观察，跟踪到函数f()使用第一个参数的位置，可以看到寄存器EBP指向栈底，图中使用红色箭头标识。栈帧中第一个被保存的是EBP的值，第二个是返回地址（RA），第三个是参数1，接下来是参数2，以此类推。因此，当我们访问第一个参数时EBP应该加8（2个32-bit字节宽度）。

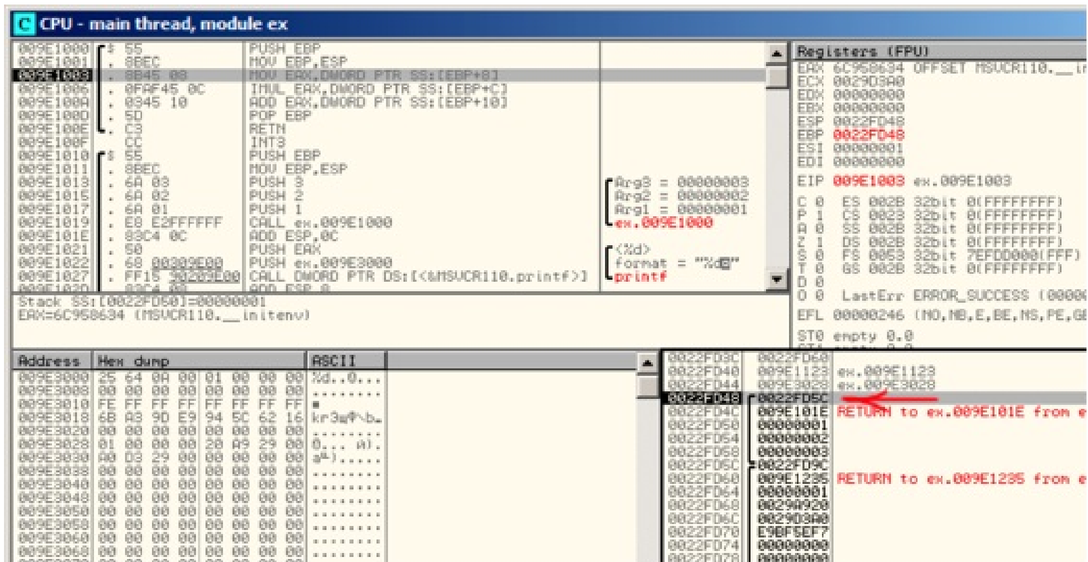


Figure 8.1: OllyDbg: 函数f()内部

8.1.3 GCC

使用GCC4.4.1编译后在IDA中查看

Listing 8.3: GCC 4.4.1

```
                public f
f               proc near

arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch
arg_8           = dword ptr 10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]  ; 1st argument
                imul    eax, [ebp+arg_4]  ; 2nd argument
                add     eax, [ebp+arg_8]  ; 3rd argument
                pop     ebp
                retn
f               endp

                public main
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_8], 3  ; 3rd argument
                mov     [esp+10h+var_C], 2  ; 2nd argument
                mov     [esp+10h+var_10], 1  ; 1st argument
                call    f
                mov     edx, offset aD ; "%d
"
                mov     [esp+10h+var_C], eax
                mov     [esp+10h+var_10], edx
                call    _printf
                mov     eax, 0
                leave
                retn
main        endp
```

几乎相同的结果。

执行两个函数后栈指针ESP并没有显示恢复，因为倒数第二个指令LEAVE（B.6.2）会自动恢复栈指针。

8.2 X64

x86-64架构下有点不同，函数参数（4或6）使用寄存器传递，被调用函数通过访问寄存器来访问传递进来的参数。

8.2.1 MSVC

MSVC优化后：

Listing 8.4: MSVC 2012 /Ox x64

```
$SG2997     DB      '%d', 0aH, 00H

main        PROC
            sub     rsp, 40
            mov     edx, 2
            lea     r8d, QWORD PTR [rdx+1]  ; R8D=3
            lea     ecx, QWORD PTR [rdx-1]  ; ECX=1
            call    f
            lea     rcx, OFFSET FLAT:$SG2997  ; '%d'
            mov     edx, eax
            call    printf
            xor     eax, eax
            add     rsp, 40
            ret     0
main        ENDP

f           PROC
            ; ECX - 1st argument
            ; EDX - 2nd argument
            ; R8D - 3rd argument
            imul    ecx, edx
            lea     eax, DWORD PTR [r8+rcx]
            ret     0
f           ENDP
```

我们可以看到函数f()直接使用寄存器来操作参数，LEA指令用来做加法，编译器认为使用LEA比使用ADD指令要更快。在mian()中也使用了LEA指令，编译器认为使用LEA比使用MOV指令效率更高。

我们来看看MSVC没有优化的情况：

Listing 8.5: MSVC 2012 x64

```
f             proc near

; shadow space:
arg_0         = dword ptr 8
arg_8         = dword ptr 10h
arg_10        = dword ptr 18h

              ; ECX - 1st argument
              ; EDX - 2nd argument
              ; R8D - 3rd argument
              mov     [rsp+arg_10], r8d
              mov     [rsp+arg_8], edx
              mov     [rsp+arg_0], ecx
              mov     eax, [rsp+arg_0]
              imul    eax, [rsp+arg_8]
              add     eax, [rsp+arg_10]
              retn
f endp

main          proc    near
              sub     rsp, 28h
              mov     r8d, 3 ; 3rd argument
              mov     edx, 2 ; 2nd argument
              mov     ecx, 1 ; 1st argument
              call    f
              mov     edx, eax
              lea     rcx, $SG2931 ; "%d
"
              call    printf

              ; return 0
              xor     eax, eax
              add     rsp, 28h
              retn
main          endp
```

这里从寄存器传递进来的3个参数因为某种情况又被保存到栈里。这就是所谓的“shadow space”2：每个Win64通常（不是必需）会保存所有4个寄存器的值。这样做由两个原因：1）为输入参数分配所有寄存器（即使是4个）太浪费，所以要通过堆栈来访问；2）每次中断下来调试器总是能够定位函数参数3。

调用者负责在栈中分配“shadow space”。

8.2.2 GCC

GCC优化后的代码：

Listing 8.6: GCC 4.4.6 -O3 x64

```
f:
        ; EDI - 1st argument
        ; ESI - 2nd argument
        ; EDX - 3rd argument
        imul    esi, edi
        lea     eax, [rdx+rsi]
        ret

main:
        sub     rsp, 8
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edi, OFFSET FLAT:.LC0 ; "%d
"
        mov     esi, eax
        xor     eax, eax ; number of vector registers passed
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

GCC无优化代码：

Listing 8.7: GCC 4.4.6 x64

```
f:
        ; EDI - 1st argument
        ; ESI - 2nd argument
        ; EDX - 3rd argument
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret

main:
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT:.LC0 ; "%d
"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0 ; number of vector registers passed
        call    printf
        mov     eax, 0
        leave
        ret
```

System V *NIX [21]没有“shadow space”，但被调用者可能会保存参数，这也是造成寄存器短缺的原因。

8.2.3 GCC: uint64_t instead int

我们例子使用的是32位int，寄存器也为32位寄存器（前缀为E-）。

为处理64位数值内部会自动调整为64位寄存器：

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
};
int main()
{
    printf ("%lld
", f(0x1122334455667788,0x1111111122222222,0x3333333344444444));
    return 0;
};
```

Listing 8.8: GCC 4.4.6 -O3 x64

```
f       proc near
        imul    rsi, rdi
        lea     rax, [rdx+rsi]
        retn
f       endp

main    proc near
        sub     rsp, 8
        mov     rdx, 3333333344444444h ; 3rd argument
        mov     rsi, 1111111122222222h ; 2nd argument
        mov     rdi, 1122334455667788h ; 1st argument
        call    f
        mov     edi, offset format ; "%lld
"
        mov     rsi, rax
        xor     eax, eax ; number of vector registers passed
        call    _printf
        xor     eax, eax
        add     rsp, 8
        retn
main    endp
```

代码非常相似，只是使用了64位寄存器（前缀为R）。

8.3 ARM

8.3.1 未优化的Keil + ARM mode

```
.text:000000A4 00 30 A0 E1              MOV     R3, R0
.text:000000A8 93 21 20 E0              MLA     R0, R3, R1, R2
.text:000000AC 1E FF 2F E1              BX      LR
...
.text:000000B0             main
.text:000000B0 10 40 2D E9              STMFD   SP!, {R4,LR}
.text:000000B4 03 20 A0 E3              MOV     R2, #3
.text:000000B8 02 10 A0 E3              MOV     R1, #2
.text:000000BC 01 00 A0 E3              MOV     R0, #1
.text:000000C0 F7 FF FF EB              BL      f
.text:000000C4 00 40 A0 E1              MOV     R4, R0
.text:000000C8 04 10 A0 E1              MOV     R1, R4
.text:000000CC 5A 0F 8F E2              ADR     R0, aD_0  ; "%d
"
.text:000000D0 E3 18 00 EB              BL      __2printf
.text:000000D4 00 00 A0 E3              MOV     R0, #0
.text:000000D8 10 80 BD E8              LDMFD   SP!, {R4,PC}
```

main()函数里调用了另外两个函数，3个值被传递到f();

正如前面提到的，ARM通常使用前四个寄存器（R0-R4）传递前四个值。

f()函数使用了前三个寄存器（R0-R2）作为参数。

MLA (Multiply Accumulate)指令将R3寄存器和R1寄存器的值相乘，然后再将乘积与R2寄存器的值相加将结果存入R0，函数返回R0。

一条指令完成乘法和加法4，如果不包括SIMD新的FMA指令5，通常x86下没有这样的指令。

第一条指令MOV R3,R0，看起来冗余是因为该代码是非优化的。

BX指令返回到LR寄存器存储的地址，处理器根据状态模式从Thumb状态转换到ARM状态，或者反之。函数f()可以被ARM代码或者Thumb代码调用，如果是Thumb代码调用BX将返回到调用函数并切换到Thumb模式，或者反之。

8.3.2 Optimizing Keil + ARM mode

```
.text:00000098             f
.text:00000098 91 20 20 E0                MLA R0, R1, R0, R2
.text:0000009C 1E FF 2F E1                BX  LR
```

这里f()编译时使用完全优化模式(-O3),MOV指令被优化，现在MLA使用所有输入寄存器并将结果置入R0寄存器。

8.3.3 Optimizing Keil + thumb mode

```
.text:0000005E 48 43                              MULS R0, R1
.text:00000060 80 18                              ADDS R0, R0, R2
.text:00000062 70 47                              BX   LR
```

Thumb模式下没有MLA指令，编译器做了两次间接处理，MULS指令使R0寄存器的值与R1寄存器的值相乘并将结果存入R0。ADDS指令将R0与R2的值相加并将结果存入R0。

8.3.4 ARM64

Optimizing GCC (Linaro) 4.9

Non-optimizing GCC (Linaro) 4.9

8.4 MIPS

第九章

一个或者多个字的返回值

X86架构下通常返回EAX寄存器的值，如果是单字节char，则只使用EAX的低8位AL。如果返回float类型则使用FPU寄存器ST(0)。ARM架构下通常返回寄存器R0。

9.1 尝试用函数的返回值返回void

假如main()函数的返回值是void而不是int会怎么样？

通常启动函数调用main()为：

```
push envp
push argv
push argc
call main
push eax
call exit
```

换句话说为

`exit(main(argc,argv,envp));`

如果main()声明为void类型并且函数没有明确返回状态值，通常在main()结束时EAX寄存器的值被返回，然后作为exit()的参数。大多数情况下函数返回的是随机值。这种情况下程序的退出代码为伪随机的。

我们看一个实例，注意main()是void类型：

```
#include <stdio.h>
void main()
{
    printf ("Hello, world!");
};
```

我们在linux下编译。

GCC 4.8.1会使用puts()替代printf()（看前面章节2.3.3），没有关系，因为puts()会返回打印的字符数，就行printf()一样。请注意，main()结束时EAX寄存器的值是非0的，这意味着main()结束时保留puts()返回时EAX的值。

Listing 9.1: GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

我们写bash脚本来看退出状态：

Listing 9.2: tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

运行：

```
$ tst.sh
Hello, world!
14
```

14为打印的字符数。

9.2 如果我们不使用返回值会发生什么？

9.3 返回一个结构体

回到返回值是EAX寄存器值的事实，这也就是为什么老的C编译器不能够创建返回信息无法拟合到一个寄存器（通常是int型）的函数。如果必须这样，应该通过指针来传递。现在可以这样，比如返回整个结构体，这种情况应该避免。如果必须要返回大的结构体，调用者必须开辟存储空间，并通过第一个参数传递指针，整个过程对程序是透明的。像手动通过第一个参数传递指针一样，只是编译器隐藏了这个过程。

小例子：

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;
    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};
```

…我们可以得到(MSVC 2010 /Ox):

```
$T3853 = 8                  ; size = 4
_a$ = 12                    ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC      ; get_some_values
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     eax, DWORD PTR $T3853[esp-4]
    lea     edx, DWORD PTR [ecx+1]
    mov     DWORD PTR [eax], edx
    lea     edx, DWORD PTR [ecx+2]
    add     ecx, 3
    mov     DWORD PTR [eax+4], edx
    mov     DWORD PTR [eax+8], ecx
    ret     0
?get_some_values@@YA?AUs@@H@Z ENDP      ; get_some_values
```

内部变量传递指针到结构体的宏为$T3853。

这个例子可以用C99语言扩展来重写：

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

Listing 9.3: GCC 4.8.1

```
_get_some_values proc near

ptr_to_struct   = dword ptr 4
a               = dword ptr 8
                mov     edx, [esp+a]
                mov     eax, [esp+ptr_to_struct]
                lea     ecx, [edx+1]
                mov     [eax], ecx
                lea     ecx, [edx+2]
                add     edx, 3
                mov     [eax+4], ecx
                mov     [eax+8], edx
                retn
_get_some_values endp
```

我们可以看到，函数仅仅填充调用者申请的结构体空间的相应字段。因此没有性能缺陷。

第十章

指针

指针通常被用作函数返回值(recall scanf() case (6)).例如，当函数返回两个值时。

10.1 Global variables example

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
    *sum=x+y;
    *product=x*y;
};

int sum, product;

void main()
{
    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d", sum, product);
};
```

编译后

Listing 10.1: Optimizing MSVC 2010 (/Ox /Ob0)

```
COMM        _product:DWORD
COMM        _sum:DWORD
$SG2803 DB              'sum=%d, product=%d', 0aH, 00H

_x$ = 8                                         ; size = 4
_y$ = 12                                        ; size = 4
_sum$ = 16                                      ; size = 4
_product$ = 20                                  ; size = 4
_f1         PROC
            mov     ecx, DWORD PTR _y$[esp-4]
            mov     eax, DWORD PTR _x$[esp-4]
            lea     edx, DWORD PTR [eax+ecx]
            imul    eax, ecx
            mov     ecx, DWORD PTR _product$[esp-4]
            push    esi
            mov     esi, DWORD PTR _sum$[esp]
            mov     DWORD PTR [esi], edx
            mov     DWORD PTR [ecx], eax
            pop     esi
            ret     0
_f1         ENDP

_main       PROC
            push    OFFSET _product
            push    OFFSET _sum
            push    456                         ; 000001c8H
            push    123                         ; 0000007bH
            call    _f1
            mov     eax, DWORD PTR _product
            mov     ecx, DWORD PTR _sum
            push    eax
            push    ecx
            push    OFFSET $SG2803
            call    DWORD PTR __imp__printf
            add     esp, 28                     ; 0000001cH
            xor     eax, eax
            ret     0
_main   ENDP
```

让我们在OD中查看：图9.1。首先全局变量地址被传递进f1()。我们在堆栈元素点击“数据窗口跟随”，可以看到数据段上分配两个变量的空间。这些变量被置0，因为未初始化数据（BSS1）在程序运行之前被清理为0。这些变量属于数据段，我们按Alt+M可以查看内存映射fig. 9.5.

让我们跟踪（F7）到f1()fig. 9.2.在堆栈中为456 (0x1C8) 和 123 (0x7B)，接着是两个全局变量的地址。

让我们跟踪到f1()结尾，可以看到两个全局变量存放了计算结果。

现在两个全局变量的值被加载到寄存器传递给printf(): fig. 10.4.

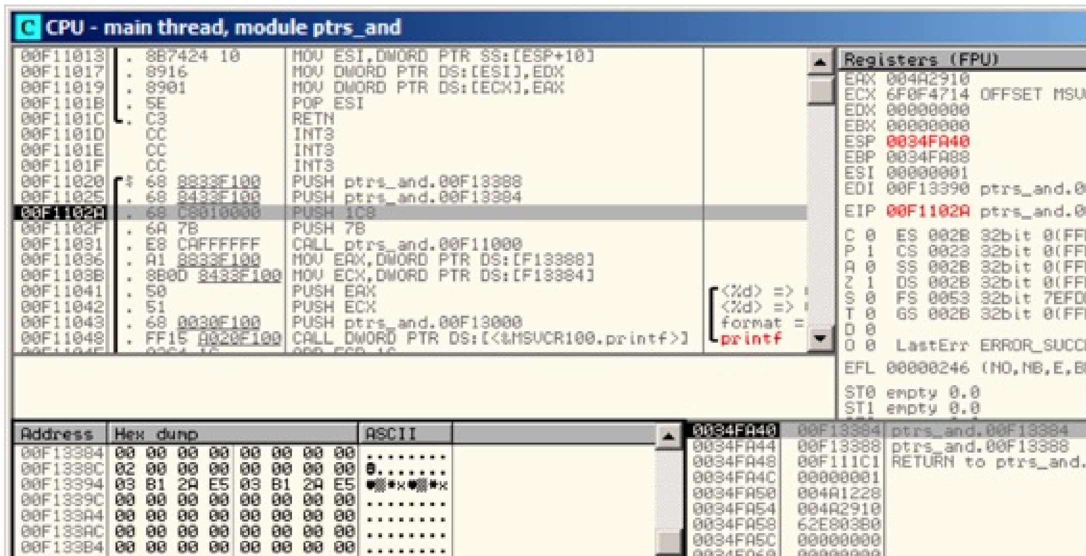


Figure 10.1: OllyDbg: 全局变量地址被传递进f1()

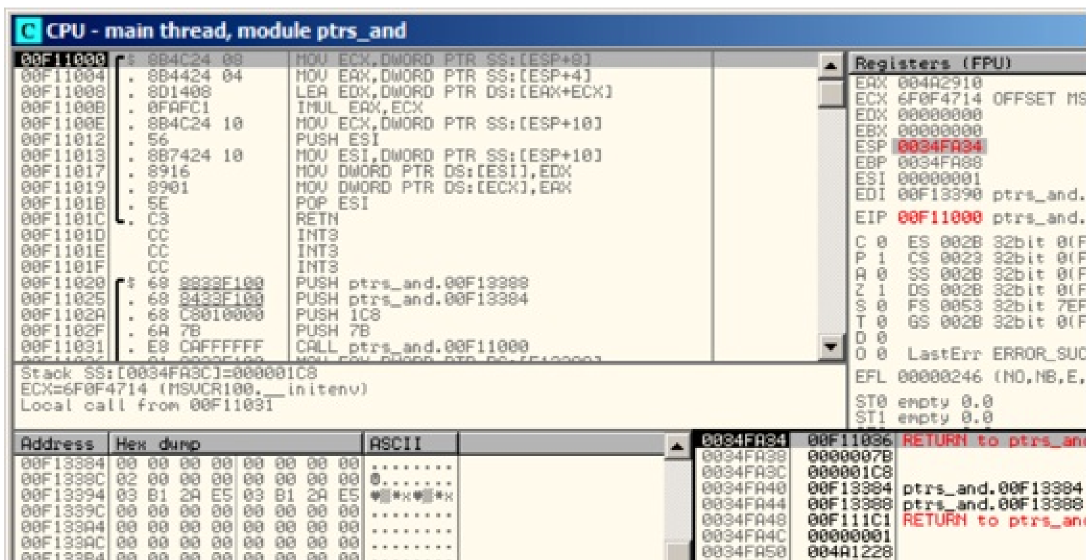


Figure 10.2: OllyDbg: f1()开始

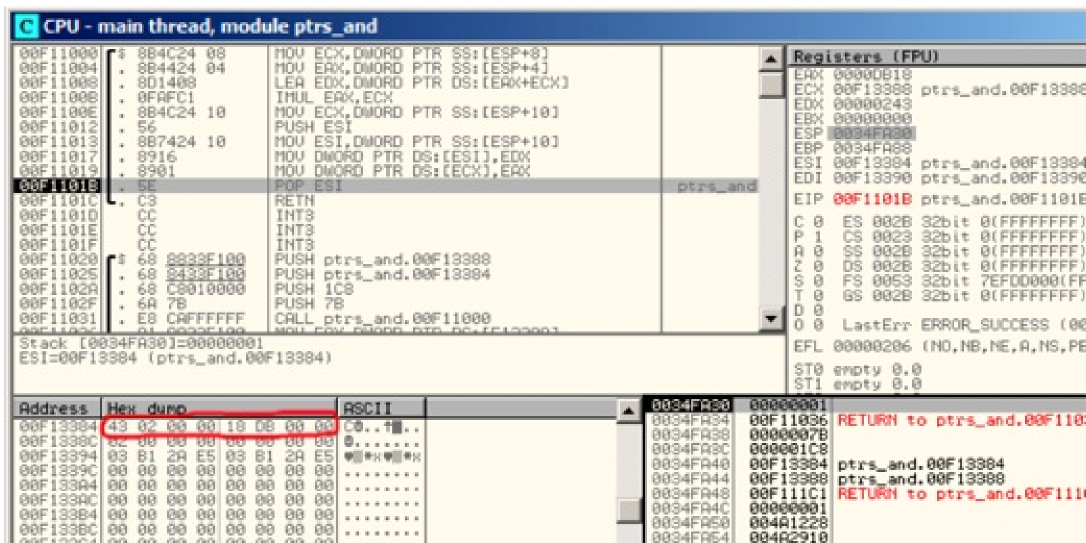


Figure 10.3: OllyDbg: f1()完成

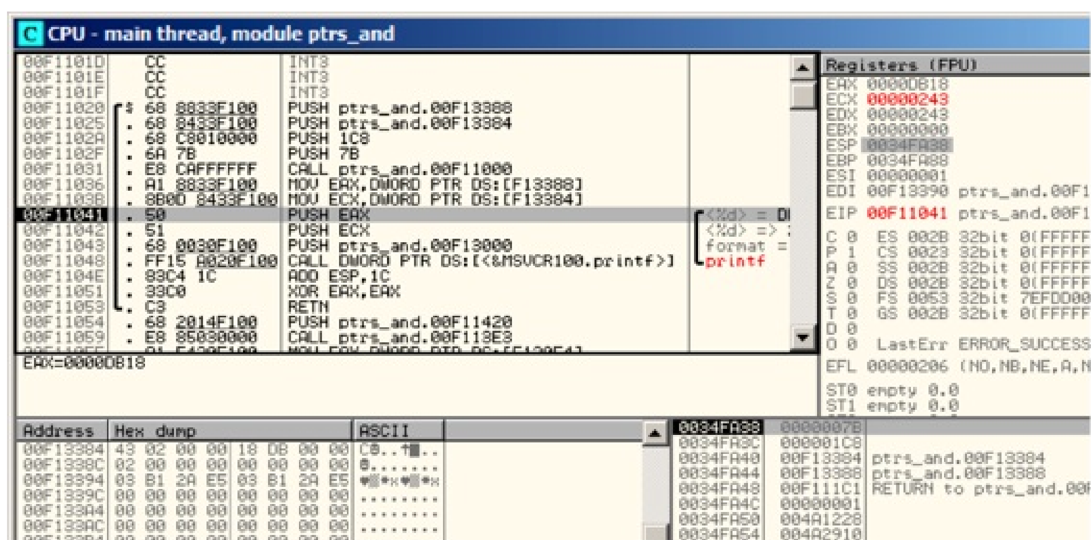


Figure 10.4: OllyDbg: 全局变量被传递进printf()

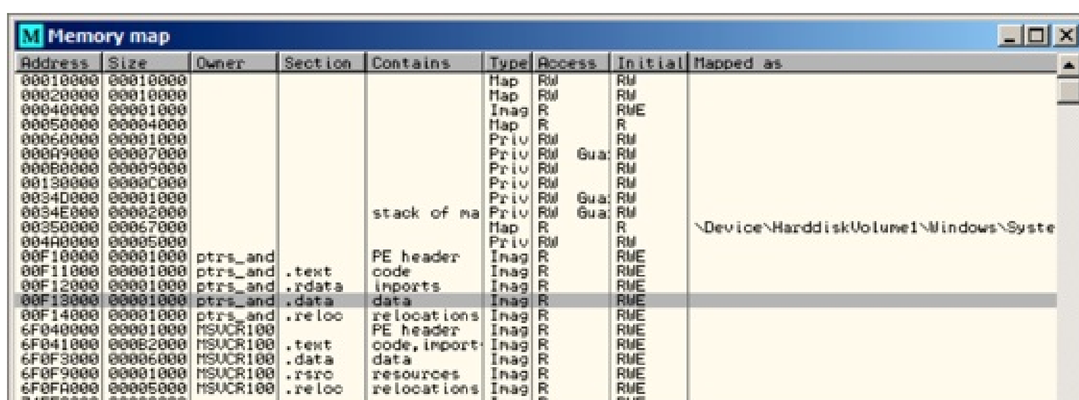


Figure 10.5: OllyDbg: memory map

10.2 Local variables example

让我们修改一下例子：

Listing 10.2: 局部变量

```
void main()
{
    int sum, product; // now variables are here

    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d
", sum, product);
};
```

f1()函数代码没有改变。仅仅main()代码作了修改。

Listing 10.3: Optimizing MSVC 2010 (/Ox /Ob0)

```
_product$ = -8                  ; size = 4
_sum$ = -4                      ; size = 4
_main    PROC
; Line 10
        sub     esp, 8
; Line 13
        lea     eax, DWORD PTR _product$[esp+8]
        push    eax
        lea     ecx, DWORD PTR _sum$[esp+12]
        push    ecx
        push    456             ; 000001c8H
        push    123             ; 0000007bH
        call    _f1
; Line 14
        mov     edx, DWORD PTR _product$[esp+24]
        mov     eax, DWORD PTR _sum$[esp+24]
        push    edx
        push    eax
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
; Line 15
        xor     eax, eax
        add     esp, 36         ; 00000024H
        ret     0
```

我们在OD中查看，局部变量地址在堆栈中是0x35FCF4和0x35FCF8。我们可以看到是如何压栈的fig. 10.6.

f1()开始的时候，随机栈地址为0x35FCF4和0x35FCF8 fig. 10.7.

f1()完成时结果0xDB18和0x243存放在地址0x35FCF4和0x35FCF8。

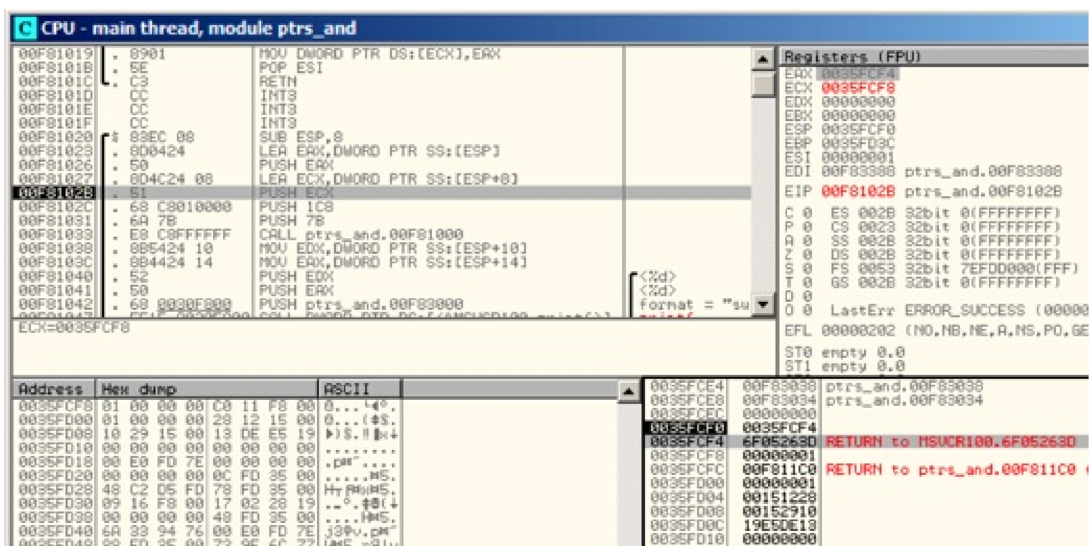


Figure 10.6: OllyDbg: 局部变量地址被压栈

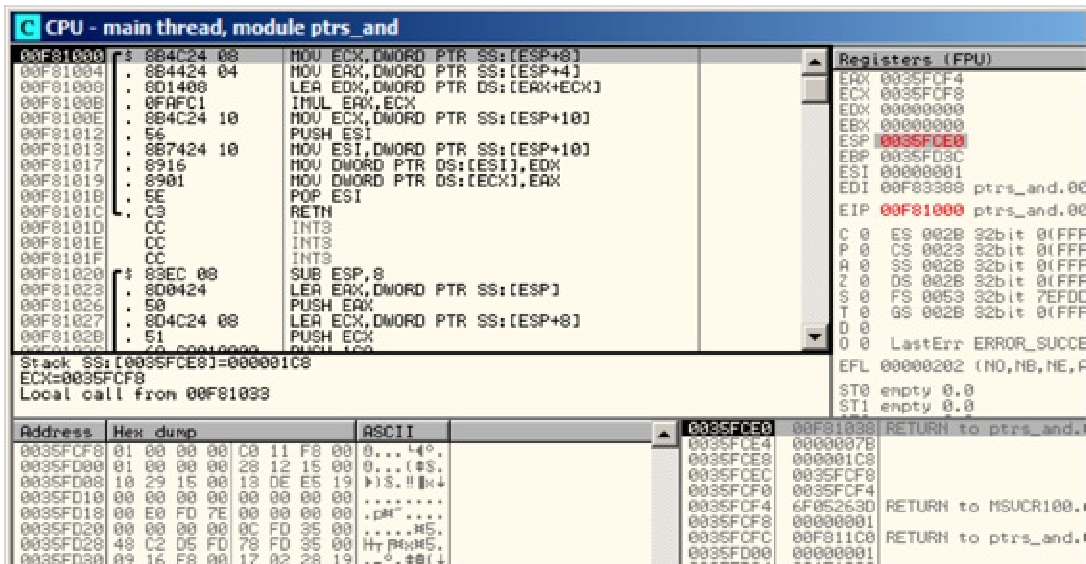


Figure 10.7: OllyDbg: f1()starting

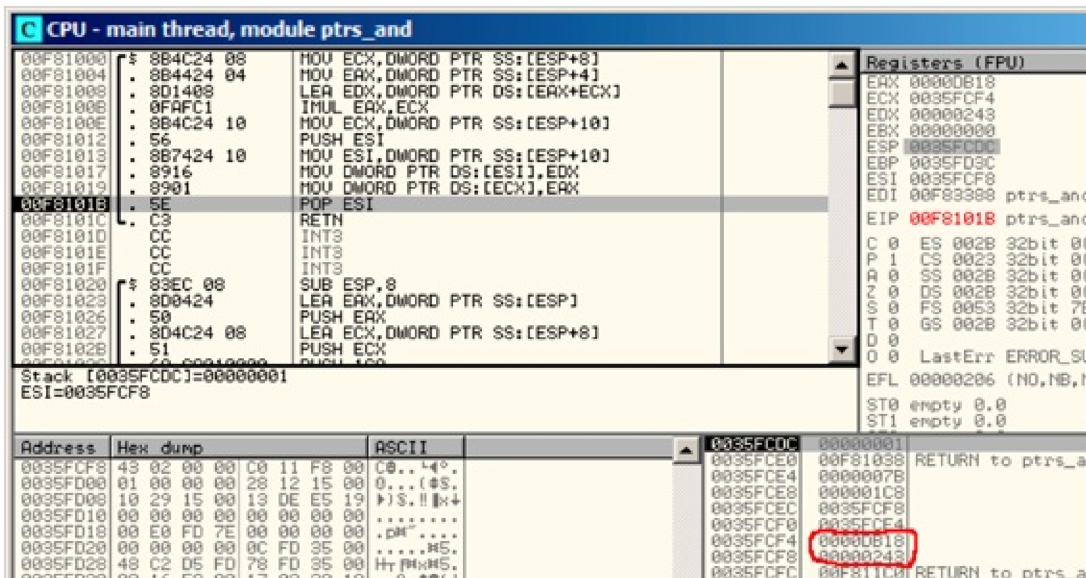


Figure 10.8: OllyDbg: f1()finished

10.3 小结

f1()可以返回结果到内存的任何地方，这是指针的本质和特性。顺便提一下，C++引用的工作方式和这个类似。详情阅读相关内容（33）。

第十一章

GOTO操作符

第十二章

条件跳转

12.1 简单的例子

现在我们来了解条件跳转。

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b");
    if (a==b)
        printf ("a==b");
    if (a<b)
        printf ("a<b");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b");
    if (a==b)
        printf ("a==b");
    if (a<b)
        printf ("a<b");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

12.1.1 x86

x86 + MSVC

f_signed() 函数:

Listing 12.1: 非优化MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed   PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        cmp     eax, DWORD PTR _b$[ebp]
        jle     SHORT $LN3@f_signed
        push    OFFSET $SG737       ; ’a>b’
        call    _printf
        add     esp, 4
$LN3@f_signed:
        mov     ecx, DWORD PTR _a$[ebp]
        cmp     ecx, DWORD PTR _b$[ebp]
        jne     SHORT $LN2@f_signed
        push    OFFSET $SG739       ; ’a==b’
        call    _printf
        add     esp, 4
$LN2@f_signed:
        mov     edx, DWORD PTR _a$[ebp]
        cmp     edx, DWORD PTR _b$[ebp]
        jge     SHORT $LN4@f_signed
        push    OFFSET $SG741       ; ’a<b’
        call    _printf
        add     esp, 4
        $LN4@f_signed:
        pop     ebp
        ret     0
_f_signed   ENDP
```

第一个指令JLE意味如果小于等于则跳转。换句话说，第二个操作数大于或者等于第一个操作数，控制流将传递到指定地址或者标签。否则（第二个操作数小于第一个操作数）第一个printf()将被调用。第二个检测JNE：如果不相等则跳转。如果两个操作数相等控制流则不变。第三个检测JGE：大于等于跳转，当第一个操作数大于或者等于第二个操作数时跳转。如果三种情况都没有发生则无printf()被调用，事实上，如果没有特殊干预，这种情况几乎不会发生。

f_unsigned()函数类似，只是JBE和JAE替代了JLE和JGE，我们来看f_unsigned()函数

Listing 12.2: GCC

```
_a$ = 8                                         ; size = 4
_b$ = 12                                        ; size = 4
_f_unsigned     PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        cmp     eax, DWORD PTR _b$[ebp]
        jbe     SHORT $LN3@f_unsigned
        push    OFFSET $SG2761 ; 'a>b'
        call    _printf
        add     esp, 4
$LN3@f_unsigned:
        mov     ecx, DWORD PTR _a$[ebp]
        cmp     ecx, DWORD PTR _b$[ebp]
        jne     SHORT $LN2@f_unsigned
        push    OFFSET $SG2763 ; 'a==b'
        call    _printf
        add     esp, 4
$LN2@f_unsigned:
        mov     edx, DWORD PTR _a$[ebp]
        cmp     edx, DWORD PTR _b$[ebp]
        jae     SHORT $LN4@f_unsigned
        push    OFFSET $SG2765 ; 'a<b'
        call    _printf
        add     esp, 4
$LN4@f_unsigned:
        pop     ebp
        ret     0
_f_unsigned     ENDP
```

几乎是相同的，不同的是：JBE-小于等于跳转和JAE-大于等于跳转。这些指令(JA/JAE/JBE/JBE)不同于JG/JGE/JL/JLE，它们使用无符号值。

我们也可以看到有符号值的表示(35)。因此我们看JG/JL代替JA/JBE的用法或者相反，我们几乎可以确定变量的有符号或者无符号类型。

main()函数没有什么新的内容：

Listing 12.3: main()

```
_main    PROC
         push     ebp
         mov      ebp, esp
         push     2
         push     1
         call     _f_signed
         add      esp, 8
         push     2
         push     1
         call     _f_unsigned
         add      esp, 8
         xor      eax, eax
         pop      ebp
         ret      0
_main    ENDP
```

10.1.2 x86 + MSVC + OllyDbg

我们在OD里允许例子来查看标志寄存器。我们从f_unsigned()函数开始。CMP执行了三次，每次的参数都相同，所以标志位也相同。

第一次比较的结果：fig. 12.1.标志位：C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. 标志位名称为OD对其的简称。

当CF=1 or ZF=1时JBE将被触发，此时将跳转。

接下来的条件跳转：fig. 12.2.当ZF=0（zero flag）时JNZ则被触发

第三个条件跳转：fig. 12.3.我们可以发现14当CF=0 (carry flag)时，JNB将被触发。在该例中条件不为真，所以第三个printf()将被执行。

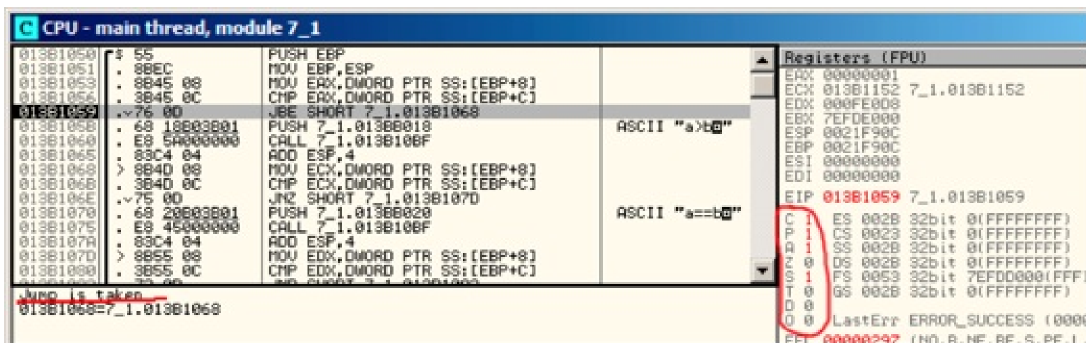


Figure 12.1: OllyDbg: f_unsigned(): 第一个条件跳转

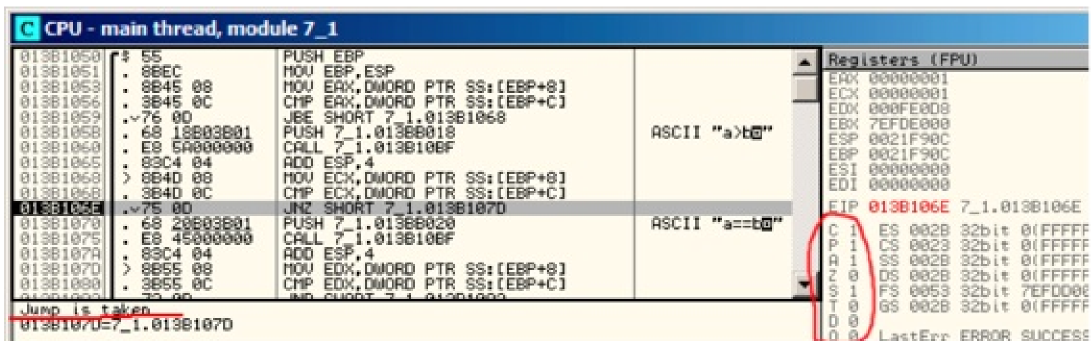


Figure 12.2: OllyDbg: f_unsigned(): 第二个条件跳转

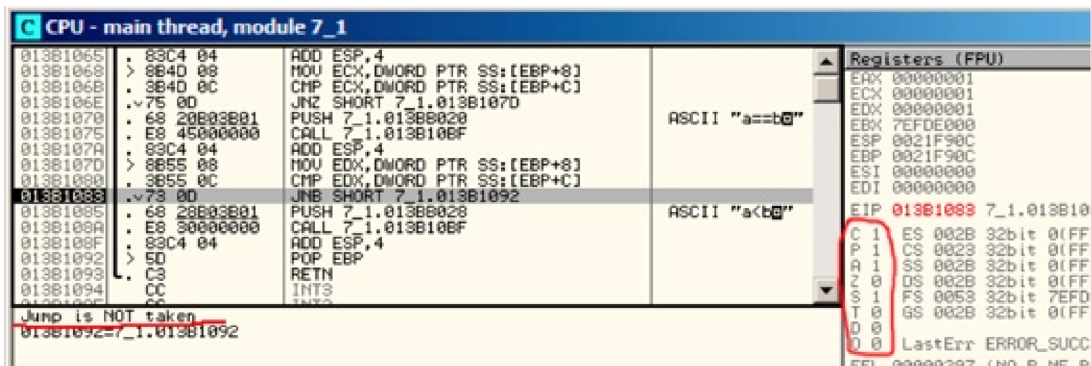


Figure 12.3: OllyDbg: f_unsigned(): 第三个条件跳转

现在我们在OD中看f_signed()函数使用有符号值。

可以看到标志寄存器：C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

第一种条件跳转JLE将被触发fig. 12.4.我们可以发现14，当ZF=1 or SF≠OF。该例中SF≠OF，所以跳转将被触发。

下一个条件跳转将被触发：如果ZF=0 (zero flag): fig. 12.5.

第三个条件跳转将不会被触发，因为仅有SF=OF，该例中不为真: fig. 12.6.

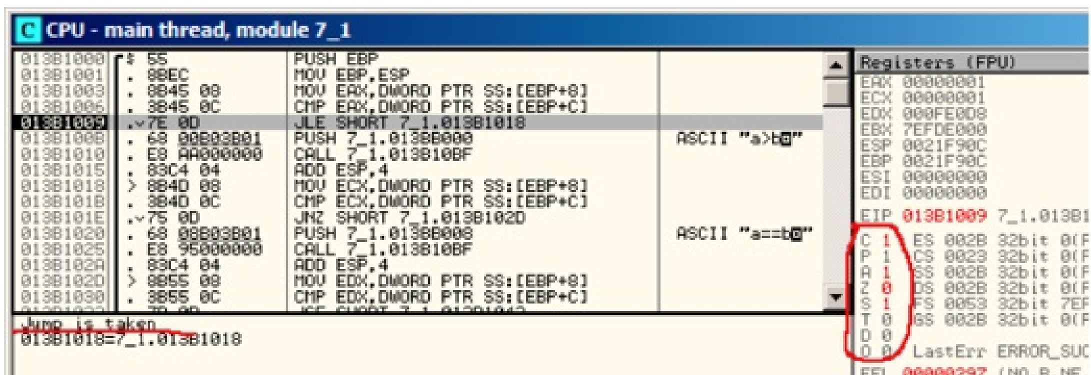


Figure 12.4: OllyDbg: f_signed(): 第一个条件跳转

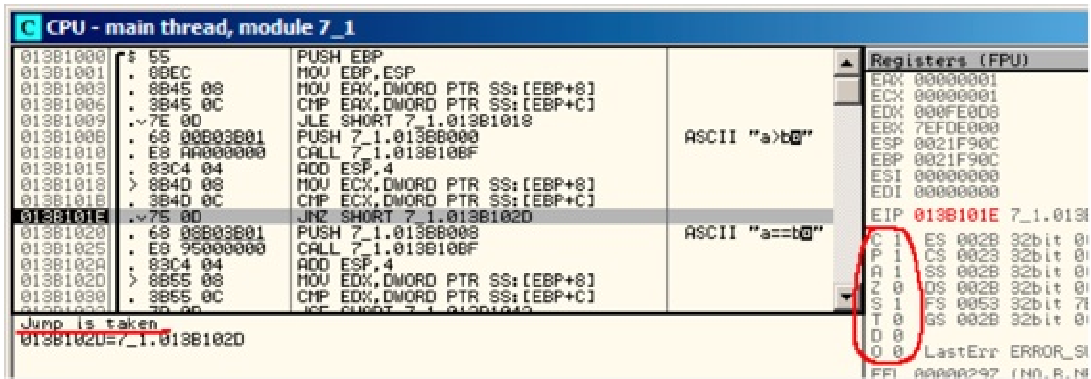


Figure 12.5: OllyDbg: f_signed(): 第二个条件跳转

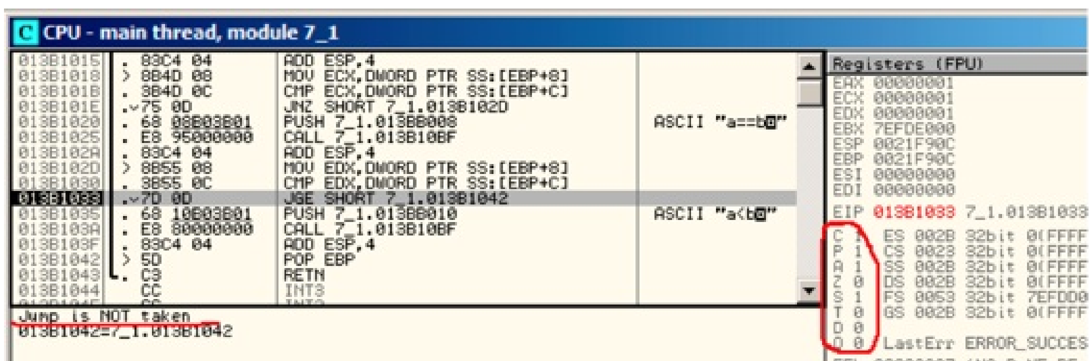


Figure 12.6: OllyDbg: f_signed(): 第三个条件跳转

x86 + MSVC + Hiew

我们可以修改这个可执行文件，使其无论输入的什么值f_unsigned()函数都会打印“a==b”。

在Hiew中查看：fig. 12.7.

我们要完成以下3个任务：

1. 使第一个跳转一直被触发；
2. 使第二个跳转从不被触发；
3. 使第三个跳转一直被触发。 我们需要使代码流进入第二个printf()，这样才一直打印“a==b”。

三个指令（或字节）应该被修改：

1. 第一个跳转修改为JMP，但跳转偏移值不变。
2. 第二个跳转有时可能被触发，我们修改跳转偏移值为0后，无论何种情况，程序总是跳向下一条指令。跳转地址等于跳转偏移值加上下一条指令地址，当跳转偏移值为0时，跳转地址就为下一条指令地址，所以无论如何下一条指令总被执行。
3. 第三个跳转我们也修改为JMP，这样跳转总被触发。 修改后：fig. 12.8.

如果忘了这些跳转，printf()可能会被多次调用，这种行为可能是我们不需要的。

```
Hiew: 7_1.exe
   C:\Polygon\ollydbg\7_1.exe        FRO --------         a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 55                        push      ebp
.00401001: 8BEC                      mov       ebp,esp
.00401003: 8B4508                    mov       eax,[ebp][8]
.00401006: 3B450C                    cmp       eax,[ebp][00C]
.00401009: 7E0D                      jle      .000401018 --1
.0040100B: 6800B04000                push      00040B000 --2
.00401010: E8AA000000                call     .0004010BF --3
.00401015: 83C404                    add       esp,4
.00401018: 8B4D08                   1mov       ecx,[ebp][8]
.0040101B: 3B4D0C                    cmp       ecx,[ebp][00C]
.0040101E: 750D                      jnz      .00040102D --4
.00401020: 6808B04000                push      00040B008 ;'a==b' --5
.00401025: E895000000                call     .0004010BF --3
.0040102A: 83C404                    add       esp,4
.0040102D: 8B5508                   4mov       edx,[ebp][8]
.00401030: 3B550C                    cmp       edx,[ebp][00C]
.00401033: 7D0D                      jge      .000401042 --6
.00401035: 6810B04000                push      00040B010 --7
.0040103A: E880000000                call     .0004010BF --3
.0040103F: 83C404                    add       esp,4
.00401042: 5D                       6pop       ebp
.00401043: C3                        retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401044: CC                        int       3
.00401045: CC                        int       3
.00401046: CC                        int       3
.00401047: CC                        int       3
.00401048: CC                        int       3
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 9lbyte 10Leave 11Naked 12AddNam
```

Figure 12.7: Hiew: f_unsigned() 函数

```
Hiew: 7_1.exe
   C:\Polygon\ollydbg\7_1.exe        FWO EDITMODE         a32 PE  00000434 Hiew 8.02 (c)SEN
 00000400: 55                        push      ebp
 00000401: 8BEC                      mov       ebp,esp
 00000403: 8B4508                    mov       eax,[ebp][8]
 00000406: 3B450C                    cmp       eax,[ebp][00C]
 00000409: EB0D                      jmps      000000418
 0000040B: 6800B04000                push      00040B000 ;' @  '
 00000410: E8AA000000                call      0000004BF
 00000415: 83C404                    add       esp,4
 00000418: 8B4D08                    mov       ecx,[ebp][8]
 0000041B: 3B4D0C                    cmp       ecx,[ebp][00C]
 0000041E: 7500                      jnz       000000420
 00000420: 6808B04000                push      00040B008 ;' @ '
 00000425: E895000000                call      0000004BF
 0000042A: 83C404                    add       esp,4
 0000042D: 8B5508                    mov       edx,[ebp][8]
 00000430: 3B550C                    cmp       edx,[ebp][00C]
 00000433: EB0D                      jmps      000000442
 00000435: 6810B04000                push      00040B010 ;' @ '
 0000043A: E880000000                call      0000004BF
 0000043F: 83C404                    add       esp,4
 00000442: 5D                        pop       ebp
 00000443: C3                        retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
 00000444: CC                        int       3
 00000445: CC                        int       3
 00000446: CC                        int       3
 00000447: CC                        int       3
 00000448: CC                        int       3
1  2NOPs 3  4  5  6  7  8Table 9  10  11  12
```

Figure 12.8: Hiew:我们修改 f_unsigned() 函数

Non-optimizing GCC

GCC 4.4.1非优化状态产生的代码几乎一样，只是用puts() (2.3.3) 替代 printf()。

12.1.5 Optimizing GCC

细心的读者可能会问，为什么要多次执行CMP，如果标志寄存器每次都相同呢？可能MSVC不会做这样的优化，但是GCC 4.8.1可以做这样的深度优化：

Listing 12.4: GCC 4.8.1 f_signed()

```
f_signed:
        mov     eax, DWORD PTR [esp+8]
        cmp     DWORD PTR [esp+4], eax
        jg      .L6
        je      .L7
        jge     .L1
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
        jmp     puts
.L6:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
        jmp     puts
.L1:
        rep     ret
.L7:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
        jmp     puts
```

我们可以看到JMP puts替代了CALL puts/RETN。稍后我们介绍这种情况11.1.1.。

不用说，这种类型的x86代码是很少见的。MSVC2012似乎不会这样做。其他情况下，汇编程序能意识到此类使用。如果你在其它地方看到此类代码，更可能是手工构造的。

f_unsigned()函数代码：

Listing 12.5: GCC 4.8.1 f_unsigned()

```
f_unsigned:
        push    esi
        push    ebx
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        mov     ebx, DWORD PTR [esp+36]
        cmp     esi, ebx
        ja      .L13
        cmp     esi, ebx ; instruction may be removed
        je      .L14
.L10:
        jb      .L15
        add     esp, 20
        pop     ebx
        pop     esi
        ret
.L15:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
.L13:
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
        call    puts
        cmp     esi, ebx
        jne     .L10
.L14:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
```

因此，GCC 4.8.1的优化算法并不总是完美的。

12.2.1ARM

32-bit ARM

Keil + ARM mode优化后

Listing 12.6: Optimizing Keil + ARM mode

```
.text:000000B8                                  EXPORT f_signed
.text:000000B8                 f_signed                      ; CODE XREF:
 main+C
.text:000000B8 70 40 2D E9                      STMFD   SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1                      MOV     R4, R1
.text:000000C0 04 00 50 E1                      CMP     R0, R4
.text:000000C4 00 50 A0 E1                      MOV     R5, R0
.text:000000C8 1A 0E 8F C2                      ADRGT   R0, aAB ; "a>b
"
.text:000000CC A1 18 00 CB                      BLGT    __2printf
.text:000000D0 04 00 55 E1                      CMP     R5, R4
.text:000000D4 67 0F 8F 02                      ADREQ   R0, aAB_0 ; "a==
b
"
.text:000000D8 9E 18 00 0B                      BLEQ    __2printf
.text:000000DC 04 00 55 E1                      CMP     R5, R4
.text:000000E0 70 80 BD A8                      LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8                      LDMFD   SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2                      ADR     R0, aAB_1 ; "a<b
"
.text:000000EC 99 18 00 EA                      B       __2printf
.text:000000EC                 ; End of function f_signed
```

ARM下很多指令只有某些标志位被设置时才会被执行。比如做数值比较时。

举个例子，ADD实施上是ADDAL，这里的AL是Always，即总被执行。判定谓词是32位ARM指令的高4位（条件域）。无条件跳转的B指令其实是有条件的，就行其它任何条件跳转一样，只是条件域为AL，这意味着总是被执行，忽略标志位。

ADRGT指令就像和ADR一样，只是该指令前面为CMP指令，并且只有前面数值大于另一个数值时（Greater Than）时才被执行。

接下来的BLGT行为和BL一样，只有比较结果符合条件才能出发（Greater Than）。ADRGT把字符串“a>b ”的地址写入R0，然后BLGT调用printf()。因此，这些指令都带有GT后缀，只有当R0（a值）大于R4（b值）时指令才会被执行。

然后我们看ADREQ和BLEQ，这些指令动作和ADR及BL一样，只有当两个操作数对比后相等时才会被执行。这些指令前面是CMP（因为printf()调用可能会修改状态标识）。 然后我们看LDMGEFD，该指令行为和LDMFD指令一样1，仅仅当第一个值大于等于另一个值时（Greater Than），指令才会被执行。

“LDMGEFD SP!, {R4-R6,PC}”恢复寄存器并返回，只是当a>=b时才被触发，这样之后函数才执行完成。但是如果a<b，触发条件不成立是将执行下一条指令LDMFD SP!, {R4-R6,LR}，该指令保存R4-R6寄存器，使用LR而不是PC，函数并不返回。最后两条指令是执行printf()（5.3.2）。

f_unsigned与此一样只是使用对应的指令为ADRHI, BLHI及LDMCSFD，判断谓词（HI = Unsigned higher, CS = Carry Set (greater than or equal)）请类比之前的说明，另外就是函数内部使用无符号数值。

我们来看一下main()函数：

Listing 12.7: main()

```
.text:00000128                                  EXPORT main
.text:00000128          main
.text:00000128 10 40 2D E9                      STMFD SP!, {R4,LR}
.text:0000012C 02 10 A0 E3                      MOV R1, #2
.text:00000130 01 00 A0 E3                      MOV R0, #1
.text:00000134 DF FF FF EB                      BL f_signed
.text:00000138 02 10 A0 E3                      MOV R1, #2
.text:0000013C 01 00 A0 E3                      MOV R0, #1
.text:00000140 EA FF FF EB                      BL f_unsigned
.text:00000144 00 00 A0 E3                      MOV R0, #0
.text:00000148 10 80 BD E8                      LDMFD SP!, {R4,PC}
.text:00000148          ; End of function main
```

这就是ARM模式如何避免使用条件跳转。

这样做有什么好处呢？因为ARM使用精简指令集（RISC）。简言之，处理器流水线技术受到跳转的影响，这也是分支预测重要的原因。程序使用的条件或者无条件跳转越少越好，使用断言指令可以减少条件跳转的使用次数。

x86没有这也的功能，通过使用CMP设置相应的标志位来触发指令。

Optimizing Keil + thumb mode

Listing 12.8: Optimizing Keil + thumb mode

```
.text:00000072      f_signed                        ; CODE XREF:
 main+6
.text:00000072 70 B5               PUSH    {R4-R6,LR}
.text:00000074 0C 00               MOVS    R4, R1
.text:00000076 05 00               MOVS    R5, R0
.text:00000078 A0 42               CMP     R0, R4
.text:0000007A 02 DD               BLE     loc_82
.text:0000007C A4 A0               ADR     R0, aAB         ; "a
>b
"
.text:0000007E 06 F0 B7 F8         BL      __2printf
.text:00000082
.text:00000082      loc_82                        ; CODE XREF: f_s
igned+8
.text:00000082 A5 42               CMP     R5, R4
.text:00000084 02 D1               BNE     loc_8C
.text:00000086 A4 A0               ADR     R0, aAB_0   ; "a==b
"
.text:00000088 06 F0 B2 F8         BL      __2printf
.text:0000008C
.text:0000008C      loc_8C                        ; CODE XREF: f_s
igned+12
.text:0000008C A5 42               CMP     R5, R4
.text:0000008E 02 DA               BGE     locret_96
.text:00000090 A3 A0               ADR     R0, aAB_1   ; "a<b
"
.text:00000092 06 F0 AD F8         BL      __2printf
.text:00000096
.text:00000096      locret_96                     ; CODE XREF: f_s
igned+1C
.text:00000096 70 BD               POP     {R4-R6,PC}
.text:00000096      ; End of function f_signed
```

仅仅Thumb模式下的B指令可能需要条件代码辅助，所以thumb代码看起来更普通一些。

BLE通常是条件跳转小于或等于（Less than or Equal），BNE—不等于（Not Equal），BGE—大于或等于（Greater than or Equal）。

f_unsigned函数是同样的，只是使用的指令用来处理无符号数值：BLS (Unsigned lower or same) 和BCS (Carry Set (Greater than or equal)).

ARM64: Optimizing GCC (Linaro) 4.9

Exercise

12.1.3 MIPS

12.2 计算绝对值

12.2.1 Optimizing MSVC

12.2.2 Optimizing Keil 6/2013: Thumb mode

12.2.3 Optimizing Keil 6/2013: ARM mode

12.2.4 Non-optimizing GCC 4.9 (ARM64)

12.2.5 MIPS

12.2.6 Branchless version?

12.3 三元操作符

12.3.1 x86

12.3.2 ARM

12.3.3 ARM64

12.3.4 MIPS

12.3.5 Let’s rewrite it in an if/else way

12.3.6 Conclusion

12.4 得到最小和最大值

12.4.1 32-bit

Branchless

12.4.2 64-bit

Branchless

12.4.3 MIPS

12.5 小结

12.5.1 x86

12.5.2 ARM

12.5.3 MIPS

12.5.4 Branchless

ARM

12.6 Exercise

第十三章

选择结构switch()/case/default

13.1 少量的 case

```
void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero"); break;
        case 1: printf ("one"); break;
        case 2: printf ("two"); break;
        default: printf ("something unknown"); break;
    };
};
```

13.1.1 X86

Non-optimizing MSVC

反汇编结果如下（MSVC 2010）：

清单13.1: MSVC 2010

```
tv64 = -4         ; size = 4
_a$ = 8           ; size = 4
_f  PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 0
    je      SHORT $LN4@f
    cmp     DWORD PTR tv64[ebp], 1
    je      SHORT $LN3@f
    cmp     DWORD PTR tv64[ebp], 2
    je      SHORT $LN2@f
    jmp     SHORT $LN1@f
$LN4@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN3@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN2@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN1@f:
    push    OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call    _printf
    add     esp, 4
$LN7@f:
    mov     esp, ebp
    pop     ebp
    ret     0
_f    ENDP
```

输出函数的switch中有一些case选择分支，事实上，它是和下面这个形式等价的：

```
void f (int a)
{
    if (a==0)
        printf ("zero");
    else if (a==1)
        printf ("one");
    else if (a==2)
        printf ("two");
    else
        printf ("something unknown");
};
```

当switch中只有少量的case分支时，我们可以看到此类代码，虽然不能确定，但是，事实上switch()在机器码级别上就是对if()的封装。这也就是说，switch()其实只是对有一大堆类似条件判断的if的一个语法糖。

在生成代码时，除了编译器把输入变量移动到一个临时本地变量tv64中之外，这块代码对我们来说并无新意。

如果是在GCC 4.4.1下编译同样的代码，我们得到的结果也几乎一样，即使你打开了最高优化（-O3）也是如此。

Optimizing MSVC

让我们在微软VC编译器中打开/Ox优化选项：`cl 1.c /Fa1.asm /Ox`

清单13.2: MSVC

```
_a$ = 8                  ; size = 4
_f  PROC
    mov     eax, DWORD PTR _a$[esp-4]
    sub     eax, 0
    je      SHORT $LN4@f
    sub     eax, 1
    je      SHORT $LN3@f
    sub     eax, 1
    je      SHORT $LN2@f
    mov     DWORD PTR _a$[esp-4], OFFSET $SG791 ; ’something unk
nown’, 0aH, 00H
    jmp     _printf
$LN2@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG789 ; ’two’, 0aH, 00
H
    jmp     _printf
$LN3@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG787 ; ’one’, 0aH, 00
H
    jmp     _printf
$LN4@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG785 ; ’zero’, 0aH, 0
0H
    jmp     _printf
_f ENDP
```

我们可以看到浏览器做了更多的难以阅读的优化（Dirty hacks）。

首先，变量的值会被放入EAX，接着EAX减0。听起来这很奇怪，但它之后是需要检查先前EAX寄存器的值是否为0的，如果是，那么程序会设置上零标志位ZF（这也表示了减去0之后，结果依然是0），第一个条件跳转语句JE（Jump if Equal 或者同义词 JZ - Jump if Zero）会因此触发跳转。如果这个条件不满足，JE没有跳转的话，输入值将减去1，之后就和之前的一样了，如果哪一次值是0，那么JE就会触发，从而跳转到对应的处理语句上。

（译注：SUB操作会重置零标志位ZF，但是MOV不会设置标志位，而JE将只有在ZF标志位设置之后才会跳转。如果需要基于EAX的值来做JE跳转的话，是需要用这个方法设置标志位的）。

并且，如果没有JE语句被触发，最终，printf()函数将收到“something unknown”的参数。

其次：我们看到了一些不寻常的东西——字符串指针被放在了变量里，然后printf()并没有通过CALL，而是通过JMP来调用的。 这个可以很简单的解释清楚，调用者把参数压栈，然后通过CALL调用函数。CALL通过把返回地址压栈，然后做无条件跳转来跳到我们的函数地址。我们的函数在执行时，不管在任何时候都有以下的栈结构（因为它没有任何移动栈指针的语句）：

```
· ESP —— 指向返回地址
· ESP+4 —— 指向变量a （也即参数）
```

另一方面，当我们这儿调用printf()函数的时候，它也需要有与我们这个函数相同的栈结构，不同之处只在于printf()的第一个参数是指向一个字符串的。 这也就是你之前看到的我们的代码所做的事情。

我们的代码把第一个参数的地址替换了，然后跳转到printf()，就像第一个没有调用我们的函数f()而是先调用了printf()一样。 printf()把一串字符输出到stdout 中，然后执行RET语句， 这一句会从栈上弹出返回地址，因此，此时控制流会返回到调用f()的函数上，而不是f()上。

这一切之所以能发生，是因为printf()在f()的末尾。在一些情况下，这有些类似于longjmp()函数。当然，这一切只是为了提高执行速度。

ARM编译器也有类似的优化，请见5.3.2节“带有多个参数的printf()函数调用”。

OllyDbg

13.1.2 ARM： 优化后的 Keil + ARM 模式

```
.text:0000014C             f1
.text:0000014C 00 00 50 E3         CMP R0, #0
.text:00000150 13 0E 8F 02         ADREQ R0, aZero     ; "zero
"
.text:00000154 05 00 00 0A         BEQ loc_170
.text:00000158 01 00 50 E3         CMP R0, #1
.text:0000015C 4B 0F 8F 02         ADREQ R0, aOne      ; "one
"
.text:00000160 02 00 00 0A         BEQ loc_170
.text:00000164 02 00 50 E3         CMP R0, #2
.text:00000168 4A 0F 8F 12         ADRNE R0, aSomethingUnkno ;
"something unknown
"
.text:0000016C 4E 0F 8F 02         ADREQ R0, aTwo      ; "two
"
.text:00000170
.text:00000170                     loc_170             ; CODE X
REF: f1+8
.text:00000170                                         ; f1+14
.text:00000170 78 18 00 EA         B __2printf
```

我们再一次看看这个代码，我们不能确定的说这就是源代码里面的switch()或者说它是if()的封装。

但是，我们可以看到这里它也在试图预测指令（像是ADREQ（相等）），这里它会在R0=0的情况下触发，并且字符串“zero”的地址将被加载到R0中。如果R0=0，下一个指令BEQ将把控制流定向到loc_170处。顺带一说，机智的读者们可能会文，之前的ADREQ已经用其他值填充了R0寄存器了，那么BEQ会被正确触发吗？答案是“是”。因为BEQ检查的是CMP所设置的标记位，但是ADREQ根本没有修改标记位。

还有，在ARM中，一些指令还会加上-S后缀，这表明指令将会根据结果设置标记位。如果没有-S的话，表明标记位并不会被修改。比如，ADD（而不是ADDS）将会把两个操作数相加，但是并不会涉及标记位。这类指令对使用CMP设置标记位之后使用标记位的指令，例如条件跳转来说非常有用。

其他指令对我们来说已经很熟悉了。这里只有一个调用指向printf（），在末尾，我们已经知道了这个小技巧（见5.3.2节）。在末尾处有三个指向printf（）的地址。还有，需要注意的是如果a=2但是a并不在它的选择分支给定的常数中时，“CMP R0, #2”指令在这个情况下就需要知道a是否等于2。如果结果为假，ADRNE将会读取字符串“something unknown ”到R0中，因为a在之前已经和0、1做过是否相等的判断了，这里我们可以假定a并不等于0或者1。并且，如果R0=2，a指向的字符串“two ”将会被ADREQ载入R0。

13.1.3 ARM：优化后的 Keil + thumb 模式

```
.text:000000D4          f1
.text:000000D4 10 B5            PUSH    {R4,LR}
.text:000000D6 00 28            CMP     R0, #0
.text:000000D8 05 D0            BEQ     zero_case
.text:000000DA 01 28            CMP     R0, #1
.text:000000DC 05 D0            BEQ     one_case
.text:000000DE 02 28            CMP     R0, #2
.text:000000E0 05 D0            BEQ     two_case
.text:000000E2 91 A0            ADR     R0, aSomethingUnkno ; "s
omething unknown
"
.text:000000E4 04 E0            B       default_case
.text:000000E6 ;
----------------------------------------------------------------
---------
.text:000000E6          zero_case                           ; CO
DE XREF: f1+4
.text:000000E6 95 A0            ADR     R0, aZero           ; "z
ero
"
.text:000000E8 02 E0            B       default_case
.text:000000EA ;
----------------------------------------------------------------
---------
.text:000000EA          one_case                            ; CO
DE XREF: f1+8
.text:000000EA 96 A0            ADR     R0, aOne            ; "o
ne
"
.text:000000EC 00 E0            B       default_case
.text:000000EE          ;
----------------------------------------------------------------
---------
.text:000000EE          two_case                            ; CO
DE XREF: f1+C
.text:000000EE 97 A0            ADR     R0, aTwo            ; "t
wo
"
.text:000000F0                  default_case                ; CO
DE XREF: f1+10
.text:000000F0                                              ; f1
+14
.text:000000F0 06 F0 7E F8      BL      __2printf
.text:000000F4 10 BD            POP     {R4,PC}
.text:000000F4           ; End of function f1
```

正如我之前提到的，在thumb模式下并没有什么功能来连接预测结果，所以这里的thumb代码有点像容易理解的x86 CISC代码。

13.1.4 ARM64: Non-optimizing GCC (Linaro) 4.9

13.1.5 ARM64: Optimizing GCC (Linaro) 4.9

13.1.6 MIPS

13.1.7 Conclusion

13.2 多case的情况

在有许多case分支的switch()语句中，对编译器来说，转换出一大堆JE/JNE语句并不是太方便。

```
void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero"); break;
        case 1: printf ("one"); break;
        case 2: printf ("two"); break;
        case 3: printf ("three"); break;
        case 4: printf ("four"); break;
        default: printf ("something unknown"); break;
    };
};
```

13.2.1 x86

反汇编结果如下（MSVC 2010）：

清单13.3: MSVC 2010

```
tv64 = -4           ; size = 4
_a$ = 8             ; size = 4
_f      PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 4
    ja      SHORT $LN1@f
    mov     ecx, DWORD PTR tv64[ebp]
    jmp     DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push    OFFSET $SG739 ; ’zero’, 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
```

```
$LN5@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN4@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN3@f:
    push    OFFSET $SG745 ; 'three', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN2@f:
    push    OFFSET $SG747 ; 'four', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN1@f:
    push    OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call    _printf
    add     esp, 4
$LN9@f:
    mov     esp, ebp
    pop     ebp
    ret     0
    npad    2
$LN11@f:
    DD  $LN6@f ; 0
    DD  $LN5@f ; 1
    DD  $LN4@f ; 2
    DD  $LN3@f ; 3
    DD  $LN2@f ; 4
_f     ENDP
```

好的，我们可以看到这儿有一组不同参数的printf()调用。 它们不仅有内存中的地址，编译器还给它们带上了符号信息。顺带一提，这些符号标签也都存在于$LN11@f内部函数表中。

在函数最开始，如果a大于4，控制流将会被传递到标签$LN1@f上，这儿会有一个参数为“something unknown”的printf()调用。

如果a值小于等于4，然后我们把它乘以4，�址的方法，这样可以正好指向我们需要的元素。比如a等于2。 那么，2×4=8（在32位进程下，所有的函数表元素的长度都只有4字节），$LN11@f的函数表地址+8——这样就能取得$LN4@f标签的位置。 JMP将从函数表中获得$LN4@f的地址，然后跳转向它。

这个函数表，有时候也叫做跳转表（jumptable）。

然后，对应的，printf()的参数就是“two”了。 字面意思， JMP DWORD PTR $LN11@f[ECX*4]* 指令意味着“跳转到存储在*$LN11@f + ecx* 4 地址上的双字”。 npad（64）是一个编译时语言宏，它用于对齐下一个标签，这样存储的地址就会按照4字节（或者16字节）对齐。这个对于处理器来说是十分合适的，因为通过内存总线、缓存从内存中获取32位的值是非常方便而且有效率的。

OllyDbg

Non-optimizing GCC

让我们看看GCC 4.4.1 生成的代码：

清单13.4： GCC 4.4.1

```
        public f
f       proc near ; CODE XREF: main+10

var_18  = dword ptr -18h
arg_0   = dword ptr 8
        push    ebp
        mov     ebp, esp
        sub     esp, 18h ; char *
        cmp     [ebp+arg_0], 4
        ja      short loc_8048444
        mov     eax, [ebp+arg_0]
        shl     eax, 2
        mov     eax, ds:off_804855C[eax]
        jmp     eax
loc_80483FE:                    ; DATA XREF: .rodata:off_804855C
        mov     [esp+18h+var_18], offset aZero ; "zero"
        call    _puts
        jmp     short locret_8048450
loc_804840C:                    ; DATA XREF: .rodata:08048560
        mov     [esp+18h+var_18], offset aOne ; "one"
        call    _puts
        jmp     short locret_8048450
loc_804841A:                    ; DATA XREF: .rodata:08048564
        mov     [esp+18h+var_18], offset aTwo ; "two"
        call    _puts
        jmp     short locret_8048450
loc_8048428:                    ; DATA XREF: .rodata:08048568
        mov     [esp+18h+var_18], offset aThree ; "three"
        call    _puts
        jmp     short locret_8048450
loc_8048436:                    ; DATA XREF: .rodata:0804856C
        mov     [esp+18h+var_18], offset aFour ; "four"
        call    _puts
        jmp     short locret_8048450
loc_8048444:                    ; CODE XREF: f+A
        mov     [esp+18h+var_18], offset aSomethingUnkno ; "some
thing unknown"
        call    _puts
locret_8048450:                 ; CODE XREF: f+26
                                ; f+34...
        leave
        retn
f       endp

off_804855C dd offset loc_80483FE ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436
```

基本和VC生成的相同，除了少许的差别：参数arg_0的乘以4操作被左移2位替换了（这集合和乘以4一样）（见17.3.1节）。然后标签地址从off_804855C处的数组获取，地址计算之后存储到EAX中，然后通过JMP EAX跳转到实际的地址上。

13.2.2 ARM：优化后的 Keil + ARM 模式

```
00000174                        f2
00000174 05 00 50 E3                    CMP     R0, #5
; switch 5 cases
00000178 00 F1 8F 30                    ADDCC   PC, PC, R0,LSL#2
; switch jump
0000017C 0E 00 00 EA                    B       default_case
; jumptable 00000178 default case
00000180                        ; --------------------------------------
-----------------------------------
00000180
00000180                        loc_180                         ; CODE X
REF: f2+4
00000180 03 00 00 EA                    B       zero_case       ; jumpta
ble 00000178 case 0
00000184                        ; --------------------------------------
-----------------------------------
00000184
00000184                        loc_184                         ; CODE X
REF: f2+4
00000184 04 00 00 EA                    B       one_case        ; jumpta
ble 00000178 case 1
00000188                        ; --------------------------------------
-----------------------------------
00000188
00000188                        loc_188                         ; CODE X
REF: f2+4
00000188 05 00 00 EA                    B       two_case        ; jumpta
ble 00000178 case 2
0000018C                        ; --------------------------------------
-----------------------------------
0000018C
0000018C                        loc_18C                         ; CODE X
REF: f2+4
0000018C 06 00 00 EA                    B       three_case      ; jumpta
ble 00000178 case 3
00000190                        ; --------------------------------------
-----------------------------------
00000190
00000190                        loc_190                         ; CODE X
REF: f2+4
00000190 07 00 00 EA                    B       four_case       ; jumpta
ble 00000178 case 4
00000194                        ; --------------------------------------
-----------------------------------
00000194
```

```
00000194                zero_case                       ; CODE X
REF: f2+4
00000194                                                ; f2:loc
_180
00000194 EC 00 8F E2            ADR     R0, aZero       ; jumpta
ble 00000178 case 0
00000198 06 00 00 EA            B       loc_1B8
0000019C                ; ---------------------------------------
-----------------------------------
0000019C
0000019C one_case                                       ; CODE X
REF: f2+4
0000019C                                                ; f2:loc
_184
0000019C EC 00 8F E2            ADR     R0, aOne        ; jumpta
ble 00000178 case 1
000001A0 04 00 00 EA            B       loc_1B8
000001A4                ; ---------------------------------------
-----------------------------------
000001A4
000001A4                two_case                        ; CODE X
REF: f2+4
000001A4                                                ; f2:loc
_188
000001A4 01 0C 8F E2            ADR     R0, aTwo        ; jumpta
ble 00000178 case 2
000001A8 02 00 00 EA            B       loc_1B8
000001AC                ; ---------------------------------------
-----------------------------------
000001AC
000001AC                three_case                      ; CODE X
REF: f2+4
000001AC                                                ; f2:loc
_18C
000001AC 01 0C 8F E2            ADR     R0, aThree ; jumptable 0
0000178 case 3
000001B0 00 00 00 EA            B       loc_1B8
000001B4 ; -----------------------------------------------------
--------------------
000001B4
000001B4                four_case                       ; CODE X
REF: f2+4
000001B4                                                ; f2:loc
_190
000001B4 01 0C 8F E2            ADR     R0, aFour       ; jumpta
ble 00000178 case 4
000001B8
000001B8                loc_1B8                         ; CODE X
REF: f2+24
000001B8                                                ; f2+2C
000001B8 66 18 00 EA            B       __2printf
000001BC ; -----------------------------------------------------
--------------------
```

```
000001BC
000001BC                default_case                        ; CODE X
REF: f2+4
000001BC                                                    ; f2+8
000001BC D4 00 8F E2            ADR     R0, aSomethingUnkno ; ju
mptable 00000178 default case
000001C0 FC FF FF EA            B       loc_1B8
000001C0                ; End of function f2
```

这个代码利用了ARM的特性，这里ARM模式下所有指令都是4个字节。

让我们记住a的最大值是4，任何更大额值都会导致它输出“something unknown”。

最开始的“CMP R0, #5”指令将a的值与5比较。

下一个“ADDCC PC, PC, R0, LSL#2”指令将仅在R0<5的时候执行（cc ==""
carry="" clear="" ，="" 小于）。所以，如果addcc并没有触发（r0="">=5时），它将会跳转到default _case标签上。

但是，如果R0<5，而且ADDCC触发了，将会发生下列事情：

R0中的值会乘以4，事实上，LSL#2代表着“左移2位”，但是像我们接下来（见17.3.1节）要看到的“移位”一样，左移2位代表乘以4。

然后，我们得到了R0 * 4的值，这个值将会和PC中现有的值相加，因此跳转到下述其中一个B（Branch 分支）指令上。

在ADDCC执行时，PC中的值（0x180）比ADDCC指令的值（0x178）提前8个字节，换句话说，提前2个指令。

这也就是为ARM处理器通道工作的方式：当ADDCC指令执行的时候，此时处理器将开始处理下一个指令，这也就是PC会指向这里的原因。

如果a=0，那么PC将不会和任何值相加，PC中实际的值将写入PC中（它相对之领先8个字节），然后跳转到标签loc_180处。这就是领先ADDCC指令8个字节的地方。

在a=1时，PC+8+a4 = PC+8+14 = PC+16= 0x184 将被写入PC中，这是loc_184标签的地址。

每当a上加1，PC都会增加4，4也是ARM模式的指令长度，而且也是B指令的长度。这组里面有5个这样的指令。

这5个B指令将传递控制流，也就是传递switch（）中指定的字符串和对应的操作等等。

13.2.3 ARM：优化后的 Keil + thumb 模式

```
000000F6                        EXPORT  f2
000000F6                f2
000000F6 10 B5                  PUSH    {R4,LR}
```

```
000000F8 03 00                  MOVS    R3, R0
000000FA 06 F0 69 F8            BL      __ARM_common_switch8_thu
mb ; switch 6 cases
000000FA                        ;--------------------------------
------------------------------------------
000000FE 05                     DCB 5
000000FF 04 06 08 0A 0C 10      DCB 4, 6, 8, 0xA, 0xC, 0x10 ; ju
mp table for switch
statement
00000105 00                     ALIGN 2
00000106
00000106                zero_case                       ; CODE X
REF: f2+4
00000106 8D A0                  ADR       R0, aZero       ; jump
table 000000FA case 0
00000108 06 E0                  B       loc_118
0000010A                        ;--------------------------------
------------------------------------------
0000010A
0000010A                    one_case                    ; CODE X
REF: f2+4
0000010A 8E A0                  ADR       R0, aOne        ; jump
table 000000FA case 1
0000010C 04 E0                  B       loc_118
0000010E                        ;--------------------------------
------------------------------------------
0000010E
0000010E                    two_case                    ; CODE X
REF: f2+4
0000010E 8F A0                  ADR       R0, aTwo        ; jump
table 000000FA case 2
00000110 02 E0                  B       loc_118
00000112                        ;--------------------------------
------------------------------------------
00000112
00000112                    three_case                  ; CODE X
REF: f2+4
00000112 90 A0                  ADR       R0, aThree      ; jump
table 000000FA case 3
00000114 00 E0                  B       loc_118
00000116                        ;--------------------------------
------------------------------------------
00000116
00000116                    four_case                   ; CODE X
REF: f2+4
00000116 91 A0                  ADR       R0, aFour       ; jump
table 000000FA case 4
00000118
00000118                    loc_118                     ; CODE X
REF: f2+12
00000118                                                ; f2+16
00000118 06 F0 6A F8            BL        __2printf
0000011C 10 BD                  POP       {R4,PC}
```

```
0000011E                        ;-------------------------------
-------------------------------------------
0000011E
0000011E                    default_case                ; CODE X
REF: f2+4
0000011E 82 A0                  ADR       R0, aSomethingUnkno ;
jumptable 000000FA default
case
00000120 FA E7                  B         loc_118

000061D0                        EXPORT __ARM_common_switch8_thum
b
000061D0                    __ARM_common_switch8_thumb ; CODE XR
EF: example6_f2+4
000061D0 78 47                  BX          PC
000061D0                    ;-----------------------------------
-----------------------------------------
000061D2 00 00                  ALIGN 4
000061D2                    ; End of function __ARM_common_switc
h8_thumb
000061D2
000061D4                        CODE32
000061D4
000061D4                    ; =============== S U B R O U T I N
E =======================================
000061D4
000061D4
000061D4                    __32__ARM_common_switch8_thumb  ; CO
DE XREF:
    __ARM_common_switch8_thumb
000061D4 01 C0 5E E5            LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1            CMP     R3, R12
000061DC 0C 30 DE 27            LDRCSB  R3, [LR,R12]
000061E0 03 30 DE 37            LDRCCB  R3, [LR,R3]
000061E4 83 C0 8E E0            ADD     R12, LR, R3,LSL#1
000061E8 1C FF 2F E1            BX      R12
000061E8                ; End of function __32__ARM_common_switc
h8_thumb
```

一个不能确定的事实是thumb、thumb-2中的所有指令都有同样的大小。甚至可以说是在这些模式下，指令的长度是可变的，就像x86一样。

所以这一定有一个特别的表单，里面包含有多少个case（除了默认的case），然后和它们的偏移，并且给他们每个都加上一个标签，这样控制流就可以传递到正确的位置。 这里有一个特别的函数来处理表单和处理控制流，被命名为__ARM_common_switch8_thumb。它由“BX PC”指令开始，这个函数用来将处理器切换到ARM模式，然后你就可以看到处理表单的函数。不过对我们来说，在这里解释它太复杂了，所以我们将省去一些细节。

但是有趣的是，这个函数使用LR寄存器作为表单的指针。还有，在这个函数调用后，LR将包含有紧跟着“BL __ARM_common_switch8_thumb”指令的地址，然后表单就由此开始。

当然，这里也不值得去把生成的代码作为单独的函数，然后再去重用它们。因此在switch()处理相似的位置、相似的case时编译器并不会生成相同的代码。

IDA成功的发觉到它是一个服务函数以及函数表，然后给各个标签加上了合适的注释，比如jumptable 000000FA case 0。

13.2.4 MIPS

13.2.5 Conclusion

当几个case在一起的时候

13.3.1 MSVC

13.3.2 GCC

13.3.3 ARM64: Optimizing GCC 4.9.1

13.4 报错

13.4.1 MSVC x86

13.4.2 ARM64

13.5 Exercises

Exercise #1

第十四章

循环结构

14.1 简单的例子

14.1.1 x86

在x86指令集中，有一些独特的LOOP指令，它们会检查ECX中的值，如果它不是0的话，它会逐渐递减ECX的值（减一），然后把控制流传递给LOOP操作符提供的标签处。也许，这个指令并不是多方便，所以，我没有看到任何现代编译器自动使用它。如果你看到哪里的代码用了这个结构，那它很有可能是程序员手写的汇编代码。

顺带一提，作为家庭作业，你可以试着解释以下为什么这个指令如此不方便。

C/C++循环操作是由for()、while()、do/while()命令发起的。

让我们从for()开始吧。

这个命令定义了循环初始值（为循环计数器设置初值），循环条件（比如，计数器是否大于一个阈值？），以及在每次迭代（增/减）时和循环体中做什么。

```
for(初始化; 条件; 每次迭代时执行的语句)
{
    循环体;
}
```

所以，它生成的代码也将被考虑为4个部分。

让我们从一个简单的例子开始吧：

```
#include <stdio.h>
void f(int i)
{
    printf ("f(%d)
", i);
};
int main()
{
    int i;
    for (i=2; i<10; i++)
    f(i);
    return 0;
};
```

反汇编结果如下（MSVC 2010）：

清单14.1: MSVC 2010

```
_i$ = -4
_main     PROC
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _i$[ebp], 2       ; loop initializatio
n
        jmp     SHORT $LN3@main
$LN2@main:
        mov     eax, DWORD PTR _i$[ebp]     ; here is what we do
 after each iteration:
        add     eax, 1                      ; add 1 to i value
        mov     DWORD PTR _i$[ebp], eax
$LN3@main:
        cmp     DWORD PTR _i$[ebp], 10      ; this condition is
checked *before* each iteration
        jge     SHORT $LN1@main             ; if i is biggest or
 equals to 10, let's finish loop
        mov     ecx, DWORD PTR _i$[ebp]     ; loop body: call f(
i)
        push    ecx
        call    _f
        add     esp, 4
        jmp     SHORT $LN2@main             ; jump to loop begin
$LN1@main:                                  ; loop end
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main ENDP
```

看起来没什么特别的。

GCC 4.4.1生成的代码也基本相同，只有一些微妙的区别。

清单14.1: GCC 4.4.1

```
main        proc near           ; DATA XREF: _start+17
var_20      = dword ptr -20h
var_4       = dword ptr -4
            push    ebp
            mov     ebp, esp
            and     esp, 0FFFFFFF0h
            sub     esp, 20h
            mov     [esp+20h+var_4], 2 ; i initializing
            jmp     short loc_8048476
loc_8048465:
            mov     eax, [esp+20h+var_4]
            mov     [esp+20h+var_20], eax
            call    f
            add     [esp+20h+var_4], 1 ; i increment
loc_8048476:
            cmp     [esp+20h+var_4], 9
            jle     short loc_8048465 ; if i<=9, continue loop
            mov     eax, 0
            leave
            retn
main        endp
```

现在，让我们看看如果我们打开了优化开关会得到什么结果（/Ox）：

清单14.3: 优化后的 MSVC

```
_main PROC
    push esi
    mov esi, 2
$LL3@main:
    push esi
    call _f
    inc esi
    add esp, 4
    cmp esi, 10 ; 0000000aH
    jl SHORT $LL3@main
    xor eax, eax
    pop esi
    ret 0
_main ENDP
```

要说它做了什么，那就是：本应在栈上分配空间的变量i被移动到了寄存器ESI里面。因为我们这样一个小函数并没有这么多的本地变量，所以它才可以这么做。 这么做的话，一个重要的条件是函数f（）不能改变ESI的值。我们的编译器在这里倒是非常确定。假设编译器决定在f（）中使用ESI寄存器的话，ESI的值将在函数的

初始化阶段被压入栈保存，并且在函数的收尾阶段将其弹出（注：即还原现场，保证程序片段执行前后某个寄存器值不变）。这个操作有点像函数开头和结束时的PUSH ESI/ POP ESI操作对。

让我们试一试开启了最高优化的GCC 4.4.1（-03优化）。

清单14.4: 优化后的GCC 4.4.1

```
main    proc near
var_10  = dword ptr -10h
        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     [esp+10h+var_10], 2
        call    f
        mov     [esp+10h+var_10], 3
        call    f
        mov     [esp+10h+var_10], 4
        call    f
        mov     [esp+10h+var_10], 5
        call    f
        mov     [esp+10h+var_10], 6
        call    f
        mov     [esp+10h+var_10], 7
        call    f
        mov     [esp+10h+var_10], 8
        call    f
        mov     [esp+10h+var_10], 9
        call    f
        xor     eax, eax
        leave
        retn
main endp
```

GCC直接把我们的循环给分解成顺序结构了。

循环分解（Loop unwinding）对这些没有太多迭代次数的循环结构来说是比较有利的，移除所有循环结构之后程序的效率会得到提升。但是，这样生成的代码明显会变得很大。

好的，现在我们把循环的最大值改为100。GCC现在生成如下：

清单14.5: GCC

```
        public main
main    proc near
var_20  = dword ptr -20h
        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        push    ebx
        mov     ebx, 2 ; i=2
        sub     esp, 1Ch
        nop     ; aligning label loc_80484D0 (loop body begin) b
y 16-byte border
loc_80484D0:
        mov     [esp+20h+var_20], ebx ; pass i as first argument
 to f()
        add     ebx, 1 ; i++
        call    f
        cmp     ebx, 64h ; i==100?
        jnz     short loc_80484D0 ; if not, continue
        add     esp, 1Ch
        xor     eax, eax ; return 0
        pop     ebx
        mov     esp, ebp
        pop     ebp
        retn
main endp
```

这时，代码看起来非常像MSVC 2010开启/Ox优化后生成的代码。除了这儿它用了EBX来存储变量i。 GCC也确信f（）函数中不会修改EBX的值，假如它要用到EBX的话，它也一样会在函数初始化和收尾时保存EBX和还原EBX，就像这里main（）函数做的事情一样。

14.1.2 OllyDbg

让我们通过/Ox和/Ob0编译程序，然后放到OllyDbg里面查看以下结果。

看起来OllyDbg能够识别简单的循环，然后把它们放在一块，为了演示方便，大家可以看图14.1。

通过跟踪代码（F8， 步过）我们可以看到ESI是如何递增的。这里的例子是ESI = i = 6： 图14.2。

9是i的最后一个循环制，这也就是为什么JL在递增的最后不会触发，之后函数结束，如图14.3。

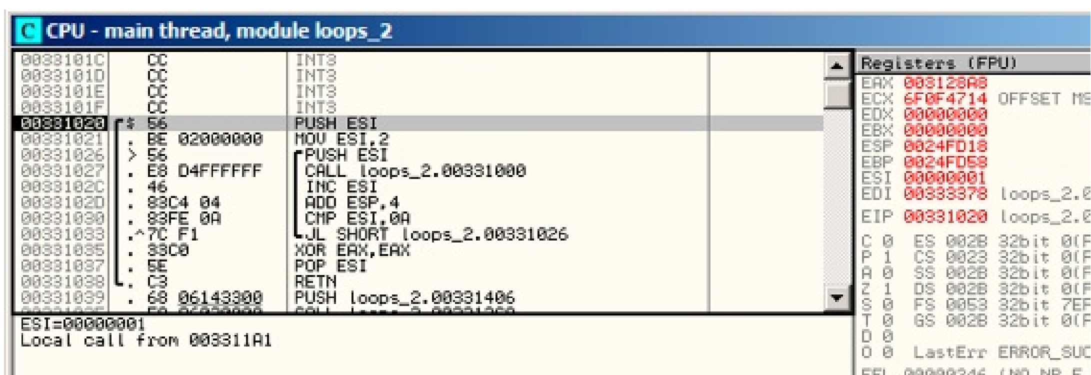


图14.1： OllyDbg main（）开始

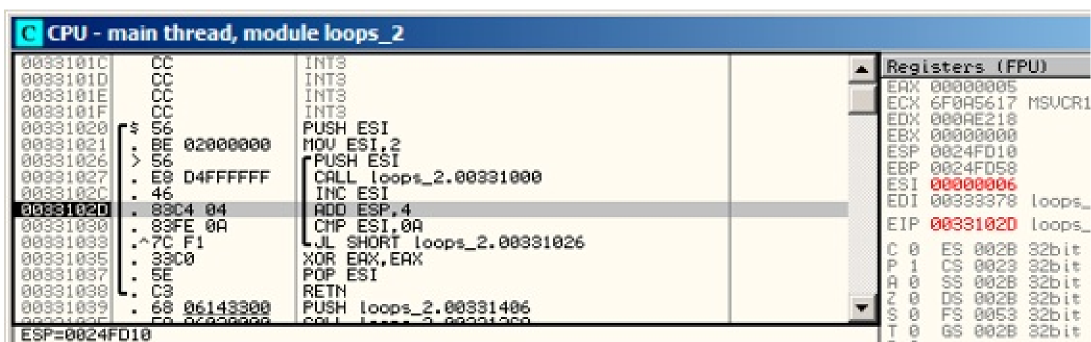


图14.2： OllyDbg： 循环体刚刚递增了i，现在i=6

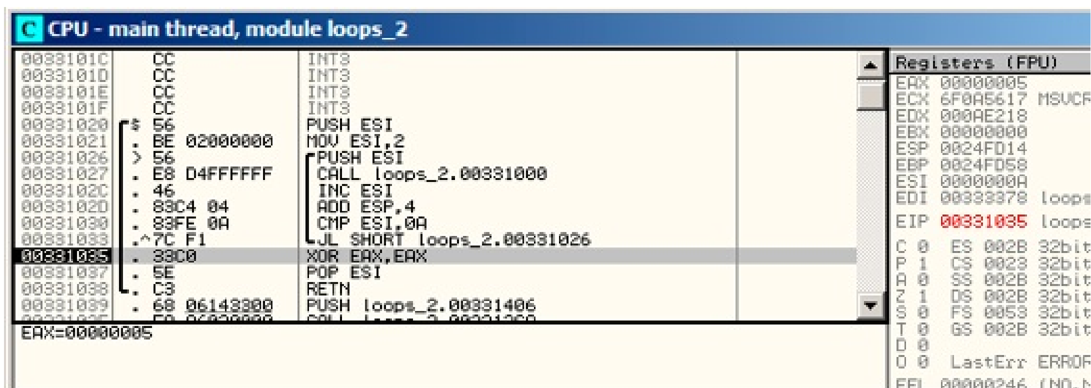


图14.3： OllyDbg中ESI=10，循环终止

14.1.2 x86:跟踪

像我们所见的一样，手动在调试器里面跟踪代码并不是一件方便的事情。这也就是我给自己写了一个跟踪程序的原因。

我在IDA中打开了编译后的例子，然后找到了PUSH ESI指令（作用：给f（）传递唯一的参数）的地址，对我的机器来说是0x401026，然后我运行了跟踪器：

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX的作用只是在对应地址上设置断点然后输出寄存器状态。

在tracer.log中我看到执行后的结果：

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

我们可以看到ESI寄存器是如何从2变为9的。

甚至于跟踪器可以收集某个函数调用内所有寄存器的值，所以它被叫做跟踪器（a trace）。每个指令都会被它跟踪上，所有感兴趣的寄存器值都会被它提示出来，然后收集下来。 然后可以生成IDA能用的.idc-script。所以，在IDA中我知道了main()函数地址是0x00401020，然后我执行了：

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

bpf的意思是在函数上设置断点。

结果是我得到了loops_2.exe.idc和loops_2.exe_clear.idc两个脚本。我加载loops_2.idc到IDA中，然后可以看到图12.4所示的内容。

我们可以看到ESI在循环体开始时从2变化为9，但是在递增完之后，它的值从9（译注：作者原文是3，但是揣测是笔误，应为9。）变为了0xA（10）。我们也可以看到main（）函数结束时EAX被设置为0。

编译器也生成了loops_2.exe.txt，包含有每个指令执行了多少次和寄存器值的一些信息：

清单14.6: loops_2.exe.txt

```
0x401020 (.text+0x20), e= 1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e= 1 [MOV ESI, 2]
0x401026 (.text+0x26), e= 8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e= 8 [CALL 8D1000h] tracing nested maximu
m level (1) reached,
skipping this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e= 8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e= 8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e= 8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e= 8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e= 1 [XOR EAX, EAX]
0x401037 (.text+0x37), e= 1 [POP ESI]
0x401038 (.text+0x38), e= 1 [RETN] EAX=0
```

生成的代码可以在此使用：

```
.text:00401020
.text:00401020 ; =============== S U B R O U T I N E =======================================
.text:00401020
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main           proc near               ; CODE XREF: ___tmainCRTStartup+11D↓p
.text:00401020
.text:00401020 argc            = dword ptr  4
.text:00401020 argv            = dword ptr  8
.text:00401020 envp            = dword ptr  0Ch
.text:00401020
.text:00401020                 push    esi             ; ESI=1
.text:00401021                 mov     esi, 2
.text:00401026
.text:00401026 loc_401026:                             ; CODE XREF: _main+13↓j
.text:00401026                 push    esi             ; ESI=2..9
.text:00401027                 call    sub_401000      ; tracing nested maximum level (1) reached,
.text:0040102C                 inc     esi             ; ESI=2..9
.text:0040102D                 add     esp, 4          ; ESP=0x38fcbc
.text:00401030                 cmp     esi, 0Ah        ; ESI=3..0xa
.text:00401033                 jl      short loc_401026 ; SF=false,true OF=false
.text:00401035                 xor     eax, eax
.text:00401037                 pop     esi
.text:00401038                 retn                    ; EAX=0
.text:00401038 _main           endp
```

图14.4： IDA加载了.idc-script之后的内容

14.1.4 ARM

无优化 Keil + ARM模式

```
main
    STMFD   SP!, {R4,LR}
    MOV     R4, #2
    B       loc_368
; ---------------------------------------------------------------
-------------

loc_35C                         ; CODE XREF: main+1C
    MOV     R0, R4
    BL      f
    ADD     R4, R4, #1
loc_368                         ; CODE XREF: main+8
    CMP     R4, #0xA
    BLT     loc_35C
    MOV     R0, #0
    LDMFD   SP!, {R4,PC}
```

迭代计数器i存储到了R4寄存器中。

```
MOV R4,#2初始化i。
MOV R0,R4 和 BL f 指令组成循环体，第一个指令为f（）准备参数，第二个用来调
用它。
ADD R4,R4, #1 指令在每次迭代中为i加一。
CMP R4,#0xA 将i和0xA（10）比较，下一个指令BLT（Branch Less Than，分支
小于）将在i<10时跳转。
否则， R0将会被写入0（因为我们的函数返回0），然后函数执行终止。
```

优化后的 Keil + ARM模式

```
_main
        PUSH    {R4,LR}
        MOVS    R4, #2

loc_132             ; CODE XREF: _main+E
        MOVS    R0, R4
        BL      example7_f
        ADDS    R4, R4, #1
        CMP     R4, #0xA
        BLT     loc_132
        MOVS    R0, #0
        POP     {R4,PC}
```

事实上，是一样的。

优化后的 Xcode（LLVM） + thumb-2 模式

```
_main
    PUSH    {R4,R7,LR}
    MOVW    R4, #0x1124 ; "%d
"
    MOVS    R1, #2
    MOVT.W  R4, #0
    ADD     R7, SP, #4
    ADD     R4, PC
    MOV     R0, R4
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #3
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #4
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #5
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #6
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #7
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #8
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #9
    BLX     _printf
    MOVS    R0, #0
    POP     {R4,R7,PC}
```

事实上，printf是在我的f（）函数里调用的：

```
void f(int i)
{
   // do something here
   printf ("%d", i);
};
```

所以，LLVM不仅仅是拆解了（unroll）循环，而且还把我的短函数f（）给作为内联函数看待了，这样，它把它的函数体内插了8遍，而不是用一个循环来解决。对于我们这种简短的函数来说，编译器这样做是有可能的。

ARM64: Optimizing GCC 4.9.1

ARM64: Non-optimizing GCC 4.9.1

14.1.5 MIPS

14.1.6 更多的一些事情

在编译器生成的代码里面，我们可以发现在i初始化之后，循环体并不会被执行，转而是先检查i的条件，在这之后才开始执行循环体。这么做是正确的，因为，如果循环条件在一开始就不满足，那么循环体是不应当被执行的。比如，在下面的例子中，就可能出现这个情况：

```
for (i=0; i<total_entries_to_process; i++)
    loop_body;
```

如果total_entries_to_process等于0，那么循环体就不应该被执行。这就是为什么应当在循环体被执行之前检查循环条件。 但是，开启编译器优化之后，如果编译器确定不会出现上面这种情况的话，那么条件检查和循环体的语句可能会互换（比如我们上面提到的简单的例子以及Keil、Xcode（LLVM）、MSVC的优化模式）。

14.2 内存块复制程序

14.2.1 直接实现

14.2.2 ARM in ARM mode

14.2.3 MIPS

14.2.4 向量化

14.3 小结

14.4 练习

第十五章

对C-Strings的简单处理

15.1 strlen()

现在，让我们再看一眼循环结构。通常，strlen()函数是由while()来实现的。这就是MSVC标准库中strlen的做法：

```
int my_strlen (const char * str)
{
    const char *eos = str;
    while( *eos++ ) ;
    return( eos - str - 1 );
}
int main()
{
    // test
    return my_strlen("hello!");
};
```

15.1 x86

无优化的 MSVC

让我们编译一下：

```
_eos$ = -4                  ; size = 4
_str$ = 8                   ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]   ; place pointer to strin
g from str
    mov     DWORD PTR _eos$[ebp], eax   ; place it to local varu
able eos
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp]   ; ECX=eos

    ; take 8-bit byte from address in ECX and place it as 32-bit
 value to EDX with sign extension

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]   ; EAX=eos
    add     eax, 1 ; increment EAX
    mov     DWORD PTR _eos$[ebp], eax   ; place EAX back to eos
    test    edx, edx                    ; EDX is zero?
    je      SHORT $LN1@strlen_          ; yes, then finish loop
    jmp     SHORT $LN2@strlen_          ; continue loop
$LN1@strlen_:

    ; here we calculate the difference between two pointers

    mov     eax, DWORD PTR _eos$[ebp]
    sub     eax, DWORD PTR _str$[ebp]
    sub     eax, 1                      ; subtract 1 and return
result
    mov     esp, ebp
    pop     ebp
    ret     0
_strlen_ ENDP
```

我们看到了两个新的指令：MOVSX（见13.1.1节）和TEST。

关于第一个：MOVSX用来从内存中取出字节然后把它放到一个32位寄存器中。MOVSX意味着MOV with Sign-Extent（带符号扩展的MOV操作）。MOVSX操作下，如果复制源是负数，从第8到第31的位将被设为1，否则将被设为0。

现在解释一下为什么要这么做。

C/C++标准将char（译注：1字节）类型定义为有符号的。如果我们有2个值，一个是char，另一个是int（int也是有符号的），而且它的初值是-2（被编码为0xFE），我们将这个值拷贝到int（译注：一般是4字节）中时，int的值将是0x000000FE，这时，int的值将是254而不是-2。因为在有符号数中，-2被编码为0xFFFFFFFE。 所以，如果我们需要将0xFE从char类型转换为int类型，那么，我们就需要识别它的符号并扩展它。这就是MOVSX所做的事情。

请参见章节“有符号数表示方法”。（35章）

我不太确定编译器是否需要将char变量存储在EDX中，它可以使用其中8位（我的意思是DL部分）。显然，编译器的寄存器分配器就是这么工作的。

然后我们可以看到TEST EDX, EDX。关于TEST指令，你可以阅读一下位这一节（17章）。但是现在我想说的是，这个TEST指令只是检查EDX的值是否等于0。

无优化的 GCC

让我们在GCC 4.4.1下测试：

```
        public strlen
strlen  proc near

eos     = dword ptr -4
arg_0   = dword ptr 8

        push    ebp
        mov     ebp, esp
        sub     esp, 10h
        mov     eax, [ebp+arg_0]
        mov     [ebp+eos], eax

loc_80483F0:
        mov     eax, [ebp+eos]
        movzx   eax, byte ptr [eax]
        test    al, al
        setnz   al
        add     [ebp+eos], 1
        test    al, al
        jnz     short loc_80483F0
        mov     edx, [ebp+eos]
        mov     eax, [ebp+arg_0]
        mov     ecx, edx
        sub     ecx, eax
        mov     eax, ecx
        sub     eax, 1
        leave
        retn
strlen  endp
```

可以看到它的结果和MSVC几乎相同，但是这儿我们可以看到它用MOVZX代替了MOVSX。 MOVZX代表着MOV with Zero-Extend（0位扩展MOV）。这个指令将8位或者16位的值拷贝到32位寄存器，然后将剩余位设置为0。事实上，这个指令比较方便的原因是它将两条指令组合到了一起：xor eax,eax / mov al, [...]。

另一方面来说，显然这里编译器可以产生如下代码： mov al, byte ptr [eax] / test al, al，这几乎是一样的，但是，EAX高位将还是会有随机的数值存在。 但是我们想一想就知道了，这正是编译器的劣势所在——它不能产生更多能让人容易理解的代

码。严格的说，事实上编译器也并没有义务为人类产生易于理解的代码。

还有一个新指令，SETNZ。这里，如果AL包含非0， test al, al将设置ZF标记位为0。 但是SETNZ中，如果ZF == 0（NZ的意思是非零，Not Zero），AL将设置为1。用自然语言描述一下，如果AL非0，我们就跳转到loc_80483F0。编译器生成了少量的冗余代码，不过不要忘了我们已经把优化给关了。

优化后的 MSVC

让我们在MSVC 2012下编译，打开优化选项/Ox：

清单15.1: MSVC 2010 /Ox /Ob0

```
_str$ = 8                      ; size = 4
_strlen     PROC
            mov     edx, DWORD PTR _str$[esp-4] ; EDX -> 指向字符的指针
            mov     eax, edx                    ; 移动到 EAX
$LL2@strlen:
            mov     cl, BYTE PTR [eax]          ; CL = *EAX
            inc     eax                         ; EAX++
            test    cl, cl                      ; CL==0?
            jne     SHORT $LL2@strlen           ; 否，继续循环
            sub     eax, edx                    ; 计算指针差异
            dec     eax                         ; 递减 EAX
            ret     0
_strlen ENDP
```

现在看起来就更简单点了。但是没有必要去说编译器能在这么小的函数里面，如此有效率的使用如此少的本地变量，特殊情况而已。

INC / DEC是递增 / 递减指令，或者换句话说，给变量加一或者减一。

优化后的 MSVC + OllyDbg

我们可以在OllyDbg中试试这个（优化过的）例子。这儿有一个简单的最初的初始化：图15.1。 我们可以看到OllyDbg

找到了一个循环，然后为了方便观看，OllyDbg把它们环绕在一个方格区域中了。在EAX上右键点击，我们可以选择“Follow in Dump”，然后内存窗口的位置将会跳转到对应位置。我们可以在内存中看到这里有一个“hello！”的字符串。 在它之后至少有一个0字节，然后就是随机的数据。 如果OllyDbg发现了一个寄存器是一个指向字符串的指针，那么它会显示这个字符串。

让我们按下F8（步过）多次，我们可以看到当前地址的游标将在循环体中回到开始的地方：图15.2。我们可以看到EAX现在包含有字符串的第二个字符。

我们继续按F8，然后执行完整个循环：图15.3。我们可以看到EAX现在包含空字符（）的地址，也就是字符串的末尾。同时，EDX并没有改变，所以它还是指向字符串的最开始的地方。现在它就可以计算这两个寄存器的差值了。

然后SUB指令会被执行：图15.4。 差值保存在EAX中，为7。 但是，字符串“hello!”的长度是6，这儿7是因为包含了末尾的。但是strlen（）函数必须返回非0部分字符串的长度，所以在最后还是要给EAX减去1，然后将它作为返回值返回，退出函数。

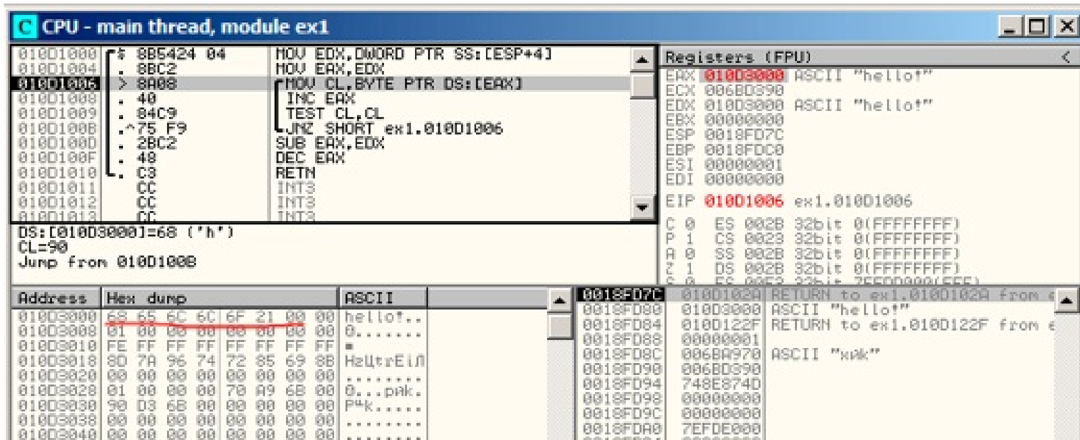


图15.1： 第一次循环迭代起始位置

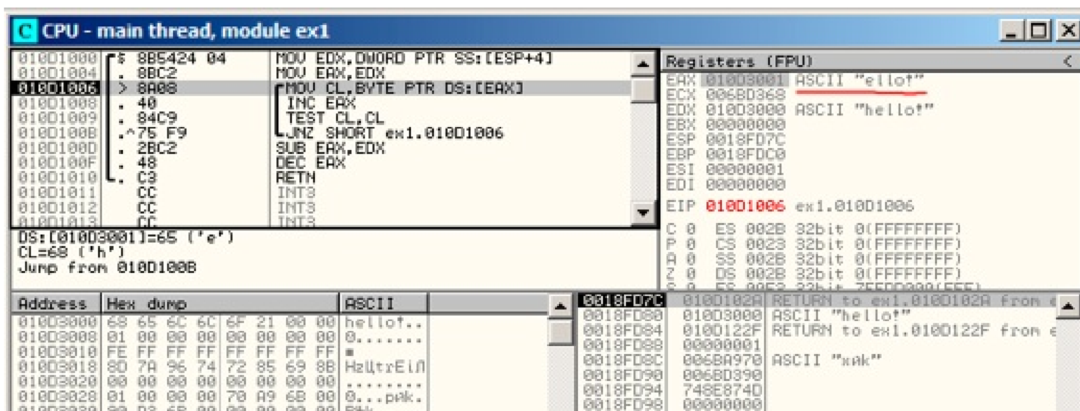


图15.2：第二次循环迭代开始位置

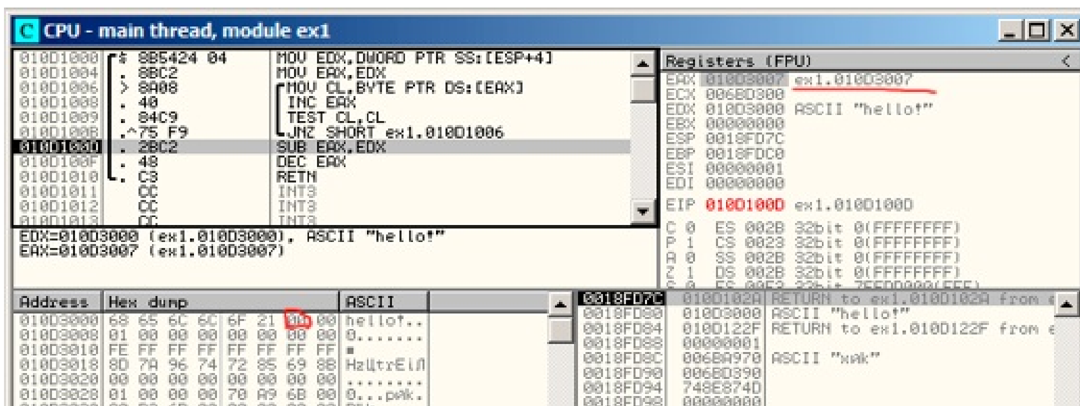


图15.3： 现在要计算二者的差了

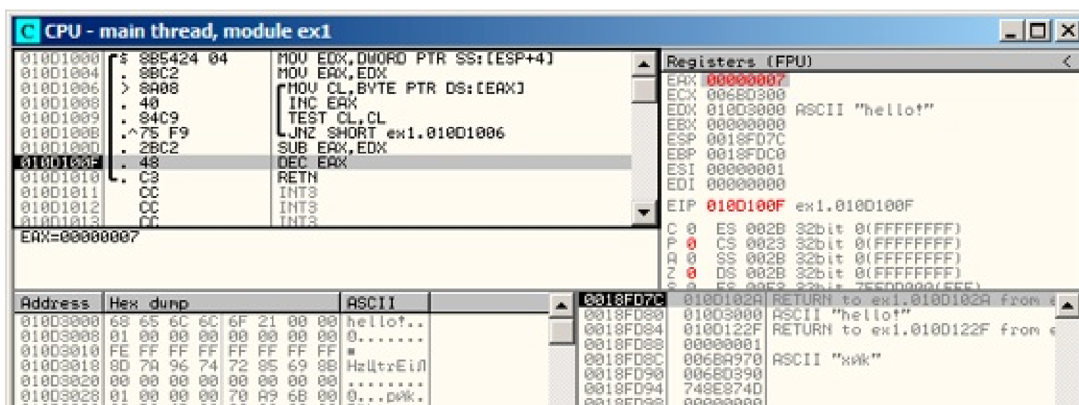


图15.4： EAX需要减一

优化过的GCC

让我们打开GCC 4.4.1的编译优化选项（-O3）：

```
        public strlen
strlen  proc near

arg_0   = dword ptr 8

        push    ebp
        mov     ebp, esp
        mov     ecx, [ebp+arg_0]
        mov     eax, ecx

loc_8048418:
        movzx   edx, byte ptr [eax]
        add     eax, 1
        test    dl, dl
        jnz     short loc_8048418
        not     ecx
        add     eax, ecx
        pop     ebp
        retn
strlen endp
```

这儿GCC和MSVC的表现方式几乎一样，除了MOVZX的表达方式。

但是，这里的MOVZX可能被替换为mov dl, byte ptr [eax]。

可能是因为对GCC编译器来说，生成此种代码会让它更容易记住整个寄存器已经分配给char变量了，然后因此它就可以确认高位在任何时候都不会有任何干扰数据的存在了。

之后，我们可以看到新的操作符NOT。这个操作符把操作数的所有位全部取反。可以说，它和XOR ECX, 0ffffffh效果是一样的。NOT和接下来的ADD指令计算差值然后将结果减一。在最开始的ECX出存储了str的指针，翻转之后会将它的值减一。

请参考“有符号数的表达方式”。（第35章）

换句话说，在函数最后，也就是循环体后面其实是做了这样一个操作：

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

这样做其实几乎相等于：

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

为什么GCC会认为它更棒呢？我不能确定，但是我确定上下两种方式都应该有相同的效率。

15.1.2 ARM

32-bit ARM

无优化 Xcode (LLVM) + ARM模式

清单15.2: 无优化的Xcode（LLVM）+ ARM模式

```
_strlen

eos     = -8
str     = -4
        SUB     SP, SP, #8 ; allocate 8 bytes for local variable
s
        STR     R0, [SP,#8+str]
        LDR     R0, [SP,#8+str]
        STR     R0, [SP,#8+eos]

loc_2CB8                ; CODE XREF: _strlen+28
        LDR     R0, [SP,#8+eos]
        ADD     R1, R0, #1
        STR     R1, [SP,#8+eos]
        LDRSB   R0, [R0]
        CMP     R0, #0
        BEQ     loc_2CD4
        B       loc_2CB8
; ----------------------------------------------------------------
--

loc_2CD4                ; CODE XREF: _strlen+24
        LDR     R0, [SP,#8+eos]
        LDR     R1, [SP,#8+str]
        SUB     R0, R0, R1 ; R0=eos-str
        SUB     R0, R0, #1 ; R0=R0-1
        ADD     SP, SP, #8 ; deallocate 8 bytes for local variab
les
        BX      LR
```

无优化的LLVM生成了太多的代码，但是，这里我们可以看到函数是如何在栈上处理本地变量的。我们的函数里只有两个本地变量，eos和str。

在这个IDA生成的列表里，我把var_8和var_4命名为了eos和str。

所以，第一个指令只是把输入的值放到str和eos里。

循环体从loc_2CB8标签处开始。

循环体的前三个指令（LDR、ADD、STR）将eos的值载入R0，然后值会加一，然后存回栈上本地变量eos。

下一条指令“LDRSB R0, [R0]”（Load Register Signed Byte，读取寄存器有符号字）将从R0地址处读取一个字节，然后把它符号扩展到32位。这有点像是x86里的MOVSX函数（见13.1.1节）。因为char在C标准里面是有符号的，所以编译器也把这个字节当作有符号数。我已经在13.1.1节写了这个，虽然那里是相对x86来说的。需要注意的是，在ARM里会单独分割使用8位或者16位或者32位的寄存器，就像x86一样。显然，这是因为x86有一个漫长的历史上的兼容性问题，它需要和他的前身：16位8086处理器甚至8位的8080处理器相兼容。但是ARM确是从32位的精简指令集处理器中发展而成的。因此，为了处理单独的字节，程序必须使用32位的寄存器。 所以LDRSB一个接一个的将符号从字符串内载入R0，下一个CMP和BEQ指

令将检查是否读入的符号是0，如果不是0，控制流将重新回到循环体，如果是0，那么循环结束。 在函数最后，程序会计算eos和str的差，然后减一，返回值通过R0返回。

注意：这个函数并没有保存寄存器。这是因为由ARM调用时的转换，R0-R3寄存器是“临时寄存器”（scratch register），它们只是为了传递参数用的，它们的值并不会在函数退出后保存，因为这时候函数也不会再使用它们。因此，它们可以被我们用来做任何事情，而这里其他寄存器都没有使用到，这也就是为什么我们的栈上事实上什么都没有的原因。因此，控制流可以通过简单跳转（BX）来返回调用的函数，地址存在LR寄存器中。

优化后的 **Xcode (LLVM) + thumb** 模式

清单13.3: 优化后的 Xcode（LLVM） + thumb模式

```
_strlen
        MOV     R1, R0

loc_2DF6                ; CODE XREF: _strlen+8
        LDRB.W  R2, [R1],#1
        CMP     R2, #0
        BNE     loc_2DF6
        MVNS    R0, R0
        ADD     R0, R1
        BX      LR
```

在优化后的LLVM中，为eos和str准备的栈上空间可能并不会分配，因为这些变量可以永远正确的存储在寄存器中。在循环体开始之前，str将一直存储在R0中，eos在R1中。

`"LDRB.W R2, [R1],#1"` 指令从R1内存中读取字节到R2里，按符号扩展成32位的值，但是不仅仅这样。 在指令最后的#1被称为“后变址”（Post-indexed address），这代表着在字节读取之后，R1将会加一。这个在读取数组时特别方便。

在x86中这里并没有这样的地址存取方式，但是在其他处理器中却是有的，甚至在PDP-11里也有。这是PDP-11中一个前增、后增、前减、后减的例子。这个很像是C语言（它是在PDP-11上开发的）中“罪恶的”语句形式ptr++、++ptr、ptr--、--ptr。顺带一提，C的这个语法真的很难让人记住。下为具体叙述：

C形式	ARM形式	C语法	工作方式
后增	后变址	*ptr++	使用*ptr 然后ptr加一
后减	后变址	*ptr--	使用*ptr 然后ptr减一
前增	前变址	*++ptr	ptr加一 然后使用*ptr
前减	前变址	*--ptr	ptr减一 然后使用*ptr

C语言作者之一的Dennis Ritchie提到了这个可能是由于另一个作者Ken Thompson开发的功能，因此这个处理器特性在PDP-7中最早出现了（参考资料[28][29]）。因此，C语言编译器将在处理器支持这种指令时使用它。

然后可以指出的是循环体的CMP和BNE，这两个指令将一直处理到字符串中的0出现为止。

MVNS（翻转所有位，也即x86的NOT）指令和ADD指令计算cos-str-1.事实上，这两个指令计算出R0=str+cos。这和源码里的指令效果一样，为什么他要这么做的原因我在13.1.5节已经说过了。

显然，LLVM，就像是GCC一样，会把代码变得更短或者更快。

优化后的 Keil + ARM 模式

清单15.4: 优化后的 Keil + ARM模式

```
_strlen
        MOV     R1, R0
loc_2C8                  ; CODE XREF: _strlen+14
        LDRB    R2, [R1],#1
        CMP     R2, #0
        SUBEQ   R0, R1, R0
        SUBEQ   R0, R0, #1
        BNE     loc_2C8
        BX      LR
```

这个和我们之前看到的几乎一样，除了str-cos-1这个表达式并不在函数末尾计算，而是被调到了循环体中间。 可以回忆一下-EQ后缀，这个代表指令仅仅会在CMP执行之前的语句互相相等时才会执行。因此，如果R0的值是0，两个SUBEQ指令都会执行，然后结果会保存在R0寄存器中。

ARM64

Optimizing GCC (Linaro) 4.9

Non-optimizing GCC (Linaro) 4.9

15.1.3 MIPS

第十六章

用其他东西代替算术指令

16.1 乘法

16.1.1 用加法代替乘法

16.2 除法

16.2.1 用移位代替除法

16.3 练习

第十七章

浮点单元

FPU是一个主cpu被设计用来处理浮点数的设备。

过去它被称为协处理器，放在CPU旁边，看起来像可编程的计算器，在学习FPU之前学习堆栈机或forth语言是值得的。

17.1 IEEE 754

x86

有趣的是，在过去（80486cpu之前），协处理器是一个单独的芯片，并不总是安装在母版上，单独购买和安装也是可以的。

但从80486 DX CPU开始,FPU就被安装在里面了。

FWAIT指令可能提醒我们一个事实——它将CPU转换成等待模式，因此它可以一直等待直到FPU完成工作。另外一点是FPU指令操作码从所谓的escape操作码（D8..DF）开始，进入了FPU。

FPU有可以容纳8个80字节的寄存器栈容量，每一个寄存器可以存储一个IEEE 754格式的数字。

C/C++语言提供至少两种浮点数类型，float（单精度，32位），double类型（双精度，64位）。

GCC也支持多精度类型（扩展精度，80位），但是MSVC不支持。

在32位环境中，浮点数要求和int类型的位数相同，但是数值的表示法完全不同。

数值包括符号位，尾数（也叫做分数）和指数。

参数列表中有float和double类型的函数通过栈来获得值，如果函数返回float或者double类型的值，那么返回值将放在ST(0)寄存器中——在FPU的栈顶。

17.3 ARM,MIPS,x86/x64 SIMD

C/C++

17.5 简单实例

下面我们来研究一个简单的例子

```
double f (double a, double b)
{
    return a/3.14 + b*4.1;
}
```

17.5.1 x86

msvc

在msvc2010中编译

```
CONST SEGMENT
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
CONST ENDS
CONST SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14
CONST ENDS
_TEXT SEGMENT
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f  PROC
    push    ebp
    mov     ebp, esp
    fld     QWORD PTR _a$[ebp]

; current stack state: ST(0) = _a

    fdiv    QWORD PTR __real@40091eb851eb851f

; current stack state: ST(0) = result of _a divided by 3.13

    fld     QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b; ST(1) = result of _a divided
by 3.13

    fmul    QWORD PTR __real@4010666666666666

; current stack state: ST(0) = result of _b * 4.1; ST(1) = resul
t of _a divided by 3.13

    faddp   ST(1), ST(0)

; current stack state: ST(0) = result of addition

    pop     ebp
    ret     0
_f ENDP
```

FLD从栈中取8个字节并将这个数字放入ST(0)寄存器中，自动将它转换成内部80位格式的扩展操作数。

FDIV除存储在ST(0)中地址指向的数值 __real@40091eb851eb851f —3.14 就放在那里。

汇编语法丢失浮点数，因此，我们这里看到的是64位IEEE754编码的16进制表示的3.14。

执行FDIV执行后，ST(0)将保存除法的结果。

另外，这里也有FDIVP指令，用ST(0)除ST(1)，从栈中将将这些值抛出来，然后将结果压栈。如果你懂forth语言，你会很快意识到这是堆栈机。

FLD指令将b的值压入栈中之后，商放入ST(1)寄存器中，ST(0)中保存b的值。

接下来FMUL指令将来自ST(0)的b值和在__real@4010666666666666 (4.1 的值在那里)相乘，然后将结果放入ST（0）中。

最后，FADDP指令将栈顶的两个值相加，将结果存储在ST(1)寄存器中，然后从ST（1）中弹出，再放入ST(0)中。

这个函数必须返回ST(0)寄存器中的值，因此，在执行FADDP命令后，没有其他额外的的指令了需要执行了。

GCC 4.4.1（选项03）生成基本同样的代码，有小小的不同之处。

不同之处在于，首先，3.14被压入栈中（进入ST(0)）,然后arg_0的值除以ST(0)寄存器中的值

FDIVR 意味着逆向除法 被除数和除数交换。

因为乘法两个乘数可交换，所以没有这样的指令，我们只有FMUL而没有逆乘。

FADDP也是将两个值相加，其中一个来自栈。然后ST(0)保存它们的和。

这段反编译代码的碎片是由IDA产生的，ST(0)简称为ST。

MSVC + OllyDbg

GCC

17.5.2 ARM: Xcode优化模式(LLVM)+ARM 模式

直到ARM有标准化的浮点数支持后，几家处理器厂商才将其加入到他们自己指令扩展中。然后，VFP（向量浮点运算单元）标准化了。

与x86相比，一个重要的不同是，在x86中使用fpu栈工作，而在ARM中，这里没有栈，你只能使用寄存器。

```
f
        VLDR        D16, =3.14
        VMOV        D17, R0, R1 ; load a
        VMOV        D18, R2, R3 ; load b
        VDIV.F64    D16, D17, D16 ; a/3.14
        VLDR        D17, =4.1
        VMUL.F64    D17, D18, D17 ; b*4.1
        VADD.F64    D16, D17, D16 ; +
        VMOV        R0, R1, D16
        BX          LR
dbl_2C98    DCFD 3.14 ; DATA XREF: f
dbl_2CA0    DCFD 4.1 ; DATA XREF: f+10
```

可以看到，这里我们使用了新的寄存器，并以D开头。这些是64位寄存器，有32个，他们既可以用作浮点数(double)运算也可以用作SIMD(在ARM中称为NEON)。

它们同时也可以作为32个32位的S寄存器使用，它们被用于单精度操作浮点数（float）运算。

记住它们很容易：D系列寄存器用于双精度数字，S寄存器用于单精度数字，记住Double和Single的首字母就可以了。

两个常量（3.14和4.1）都是以IEEE 754的形式存储在内存中。

VLDR和VMOV指令，容易推断，类似LDR和MOV指令，但是它们使用D系列寄存器，需要注意的就是这些指令不就之后也会展现出，就像D系列寄存器一样，不仅可以进行浮点数运算而且也可以用于SIMD(NEON)运算，参数传递的方式仍旧是通过R系列寄存器传递，但是每个具有双精度的数值有64位，所以为了便于传递需要两个寄存器。

```
VMOV D17,R0,R1在最开始，将两个来自R0和R1的32位的值组成一个64位的值并且将它保存在D17中。
VMOV R0,R1,D16是一个逆操作，D16中的值放回R0,R1中。
VDIV,VMUL,VADD都是用于浮点数的处理计算的指令，分别为除法指令，乘法指令，加法指令。
```

thumb-2的代码也是相同的。

17.5.3 ARM:优化 keil＋thumb 模式

```
f
            PUSH    {R3-R7,LR}
            MOVS    R7, R2
            MOVS    R4, R3
            MOVS    R5, R0
            MOVS    R6, R1
            LDR     R2, =0x66666666
            LDR     R3, =0x40106666
            MOVS    R0, R7
            MOVS    R1, R4
            BL      __aeabi_dmul
            MOVS    R7, R0
            MOVS    R4, R1
            LDR     R2, =0x51EB851F
            LDR     R3, =0x40091EB8
            MOVS    R0, R5
            MOVS    R1, R6
            BL      __aeabi_ddiv
            MOVS    R2, R7
            MOVS    R3, R4
            BL      __aeabi_dadd
            POP     {R3-R7,PC}
dword_364   DCD 0x66666666          ; DATA XREF: f+A
dword_368   DCD 0x40106666          ; DATA XREF: f+C
dword_36C   DCD 0x51EB851F          ; DATA XREF: f+1A
dword_370   DCD 0x40091EB8          ; DATA XREF: f+1C
```

keil为处理器生成的代码不支持FPU和NEON。因此，双精度浮点数通过通用R寄存器来传递双精度数字，与FPU指令不同的是，通过对库函数调用（如**aeabi_dmul,** aeabi_ddiv, __aeabi_dadd）用来实现乘法，除法，浮点数加法。当然，这比FPU协处理器慢，但总比没有强。

另外，在x86的世界中，当协处理器少而贵并且只安装昂贵的计算机上时，在FPU模拟库非常受欢迎。

在ARM的世界中，FPU处理器模拟称为soft float 或者armel，用协处理器的FPU指令的称为hard float和armhf。

举个例子，树莓派的linux内核用两种变量编译。如果是soft float，参数就会通过R系列寄存器编码，hard float则会通过D系列寄存器。

这就是不让你使用例子中来自armel编码的armhf库原因，反之亦然。那也是linux分区必须根据调用惯例编译的原因。

17.5.4 ARM64: Optimizing GCC (Linaro) 4.9

17.5.5 ARM64: Non-optimizing GCC (Linaro) 4.9

17.5.6 MIPS

17.6 通过参数通过浮点数

```
#include <math.h>
#include <stdio.h>
int main ()
{
    printf ("32.01 ^ 1.54 = %lf", pow (32.01,1.54));
    return 0;
}
```

17.6.1 x86

让我们来看看在（msvc2010）中得到的东西

清单15.3 ：MSVC 2010

```
CONST   SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r   ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r   ; 1.54
CONST ENDS

_main     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 8 ; allocate place for the first variable
        fld     QWORD PTR __real@3ff8a3d70a3d70a4
        fstp    QWORD PTR [esp]
        sub     esp, 8 ; allocate place for the second variable
        fld     QWORD PTR __real@40400147ae147ae1
        fstp    QWORD PTR [esp]
        call    _pow
        add     esp, 8 ; "return back" place of one variable.

; in local stack here 8 bytes still reserved for us.
; result now in ST(0)

        fstp    QWORD PTR [esp] ; move result from ST(0) to loca
l stack for printf()
        push    OFFSET $SG2651
        call    _printf
        add     esp, 12
        xor     eax, eax
        pop     ebp
        ret     0
_main       ENDP
```

FLD和FSTP读取FPU的栈中的变量。pow（）从FPU栈中拿出两个值然后将结果返回到ST(0)寄存器中。printf（）函数从本地栈中取出8字节并且将他们翻译为双精度变量。

17.6.2 ARM+Non-optimizing Xcode（LLVM）+thumb-2模式

```
_main
var_C        = -0xC
             PUSH    {R7,LR}
             MOV     R7, SP
             SUB     SP, SP, #4
             VLDR    D16, =32.01
             VMOV    R0, R1, D16
             VLDR    D16, =1.54
             VMOV    R2, R3, D16
             BLX     _pow
             VMOV    D16, R0, R1
             MOV     R0, 0xFC1 ; "32.01 ^ 1.54 = %lf
"
             ADD     R0, PC
             VMOV    R1, R2, D16
             BLX     _printf
             MOVS    R1, 0
             STR     R0, [SP,#0xC+var_C]
             MOV     R0, R1
             ADD     SP, SP, #4
             POP     {R7,PC}
dbl_2F90     DCFD 32.01      ; DATA XREF: _main+6
dbl_2F98     DCFD 1.54       ; DATA XREF: _main+E
```

就像我以前写的一样，64位的浮点数是成对传递给R系列寄存器的。这样的代码是冗余的（当然是因为优化选项关掉了），因为，事实上直接从R系列寄存器传递值，不借助D系列寄存器是可能的。

因此我们可以看到，_pow 将第一个参数放入R0和R1中，第二个参数放入R2和R3中。函数结果放入R0和R1中。_pwn的结果先放入了D16中，然后再放入R1和R2中，然后printf函数将取走这个值。

17.6.3 ARM+非优化模式keil＋ARM模式

```
_main
                STMFD   SP!, {R4-R6,LR}
                LDR     R2, =0xA3D70A4 ; y
                LDR     R3, =0x3FF8A3D7
                LDR     R0, =0xAE147AE1 ; x
                LDR     R1, =0x40400147
                BL      pow
                MOV     R4, R0
                MOV     R2, R4
                MOV     R3, R1
                ADR     R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf
"
                BL      __2printf
                MOV     R0, #0
                LDMFD   SP!, {R4-R6,PC}

y               DCD 0xA3D70A4       ; DATA XREF: _main+4
dword_520       DCD 0x3FF8A3D7      ; DATA XREF: _main+8
; double x
x               DCD 0xAE147AE1      ; DATA XREF: _main+C
dword_528       DCD 0x40400147      ; DATA XREF: _main+10
a32_011_54Lf    DCB "32.01 ^ 1.54 = %lf",0xA,0
                                    ; DATA XREF: _main+24
```

D系列寄存器在这里不使用，只成对地使用R系列的寄存器

17.6.4 ARM64 + Optimizing GCC (Linaro) 4.9

17.6.5 MIPS

15.3 对比实例

试试这个

```
double d_max (double a, double b)
{
    if (a>b)
    return a;
    return b;
};
```

17.7.1 x86

无优化的MSVC

尽管这个函数很简单，但是理解它的工作原理并不容易。

MSVC 2010生成

```
PUBLIC      _d_max
_TEXT   SEGMENT
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_d_max      PROC
    push    ebp
    mov     ebp, esp
    fld     QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b
; compare _b (ST(0)) and _a, and pop register

    fcomp   QWORD PTR _a$[ebp]

; stack is empty here

    fnstsw  ax
    test    ah, 5
    jp      SHORT $LN1@d_max

; we are here only if a>b

    fld     QWORD PTR _a$[ebp]
    jmp     SHORT $LN2@d_max
$LN1@d_max:
    fld     QWORD PTR _b$[ebp]
$LN2@d_max:
    pop     ebp
    ret     0
_d_max      ENDP
```

因此，FLD将_b中的值装入ST(0)寄存器中。

FCOMP对比ST(0)寄存器和_a值，设置FPU状态字寄存器中的C3/C2/C0位，这是一个反应FPU当前状态的16位寄存器。

C3/C2/C0位被设置后，不幸的是，IntelP6之前的CPU没有任何检查这些标志位的条件转移指令。可能是历史的原因（FPU曾经是单独的一块芯片）。从Intel P6开始，现在的CPU拥有FCOMI/FCOMIP/FUCOMI/FUCOMIP指令，这些指令功能相同，但会改变CPU的ZF/PF/CF标志位。

当标志位被设好后，FCOMP指令从栈中弹出一个变量。这就是和FCOM的不同之处，FCOM只对比值，让栈保持同样的状态。

FNSTSW讲FPU状态字寄存器的内容拷贝到AX中，C3/C2/C0放置在14/10/8位中，它们会在AX寄存器中相应的位置上，并且都放在AX的高位部分—AH。

```
如果 b>a 在我们的例子中，C3/C2/C0位会被设置为：0，0，0
如果 a>b 标志位被设为:0,0,1
如果 a=b 标识位被设为：1，0，0
```

执行了 test sh，5 之后，C3和C1的标志位被设为0，但是第0位和第2位（在AH寄存器中）C0和C2位会保留。

下面我们谈谈奇偶位标志。Another notable epoch rudiment：

一个常见的原因是测试奇偶位标志事实上与奇偶没有任何关系。FPU有4个条件标志（C0到C3），但是它们不能被直接测试，必须先拷贝到标志位寄存器中，在这个时候，C0放在进位标志中，C2放在奇偶位标志中，C3放在0标志位中。当例子中不可比较的浮点数（NaN或者其他不支持的格式）使用FUCOM指令进行比较的时候，会设置C2标志位。

如果一个数字是奇数这个标志就会被设置为1。如果是偶数就会被设置为0.

因此，PF标志会被设置为1如果C0和C2都被设置为0或者都被设置为1。然后jp跳转就会实现。如果我们recall valuesof C3/C2/C0，我们将会发现条件跳转jp可能会在两种情况下触发：b>a或者a==b（C3位这里不再考虑，因为在执行test sh,5指令之后已经被清零了）

之后就简单了。如果条件跳转被触发，FLD会将_b的值放入ST(0)寄存器中，如果没有被触发，_a变量的值会被加载 但是还没有结束。

关于检查C2-Flag

First OllyDbg example: a=1.2 and b=3.4

Second OllyDbg example: a=5.6 and b=-4

msvc2010优化模式

```
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_d_max  PROC
    fld     QWORD PTR _b$[esp-4]
    fld     QWORD PTR _a$[esp-4]
; current stack state: ST(0) = _a, ST(1) = _b
    fcom ST(1) ; compare _a and ST(1) = (_b)
    fnstsw ax
    test ah, 65 ; 00000041H
    jne SHORT $LN5@d_max
    fstp ST(1) ; copy ST(0) to ST(1) and pop register, leave (_a
) on top
; current stack state: ST(0) = _a
    ret 0
$LN5@d_max:
    fstp ST(0) ; copy ST(0) to ST(0) and pop register, leave (_b
) on top
; current stack state: ST(0) = _b
    ret 0
_d_max ENDP
```

FCOM区别于FCOMP在某种程度上是它只比较值然后并不改变FPU的状态。和之前的例子不同的是，操作数是逆序的。这也是C3/C2/C0中的比较结果是不同的原因。

```
如果 a>b  在我们的例子中，C3/C3/C0会被设为0，0，0
如果 b>a  标志位被设为：0，0，1
如果 a=b  标志位被设为：1，0，0
```

可以这么说，test ah,65指令只保留两位—C3和C0.如果a>b那么两者都被设为0：在那种情况下，JNE跳转不会被触发。 FSTP ST(1）接下来—这个指令会复制ST(0)中的值放入操作数中，然后从FPU栈中跑出一个值。 换句话说，这个这个指令将ST(0)中的值复制到ST(1)中。然后，_a的两个值现在在栈定。之后，一个值被抛出。之后，ST(0)会包含_a然后函数执行完毕。

条件跳转JNE在两种情况下触发：b>a或者a==b。ST(0)中的值拷贝到ST（0）中，就像nop指令一样,然后一个值从栈中抛出，然后栈顶(ST(0）)会包含ST(1)之前的包含的内容（就是_b）。函数执行完毕。这条指令在这里使用的原因可能是FPU没有从栈中抛出值的指令并且没有地方存储。 但是，还没有结束。

First OllyDbg example: a=1.2 and b=3.4

Second OllyDbg example: a=5.6 and b=-4

GCC 4.4.1

```
d_max proc near
b               =qword ptr -10h
a               =qword ptr -8
a_first_half    = dword ptr 8
a_second_half   = dword ptr 0Ch
b_first_half    = dword ptr 10h
b_second_half   = dword ptr 14h

    push    ebp
    mov     ebp, esp
    sub     esp, 10h

; put a and b to local stack:

    mov     eax, [ebp+a_first_half]
    mov     dword ptr [ebp+a], eax
    mov     eax, [ebp+a_second_half]
    mov     dword ptr [ebp+a+4], eax
    mov     eax, [ebp+b_first_half]
    mov     dword ptr [ebp+b], eax
    mov     eax, [ebp+b_second_half]
    mov     dword ptr [ebp+b+4], eax

; load a and b to FPU stack:

    fld     [ebp+a]
    fld     [ebp+b]
; current stack state: ST(0) - b; ST(1) - a

    fxch    st(1) ; this instruction swapping ST(1) and ST(0)

; current stack state: ST(0) - a; ST(1) - b

    fucompp     ; compare a and b and pop two values from stack,
 i.e., a and b
    fnstsw  ax  ; store FPU status to AX
    sahf        ; load SF, ZF, AF, PF, and CF flags state from A
H
    setnbe  al  ; store 1 to AL if CF=0 and ZF=0
    test    al, al              ; AL==0 ?
    jz      short loc_8048453    ; yes
    fld     [ebp+a]
    jmp     short locret_8048456

loc_8048453:
    fld     [ebp+b]
locret_8048456:
    leave
    retn
d_max endp
```

FUCOMMP 类似FCOM指令，但是两个值都从栈中取，并且处理NaN(非数)有一些不同之处。

更多关于”非数“的：

FPU能够处理特殊的值比如非数字或者NaNs。它们是无穷大的，除零的结果等等。NaN可以是“quiet”并且“signaling”的。但是如果进行任何有关“signaling”的操作将会产生异常。

FCOM会产生异常如果操作数中有NaN。FUCOM只在操作数有signaling NaN (SNaN)的情况下产生异常。

接下来的指令是SANF—这条指令很少用，它不使用FPU。AH的8位以这样的顺序放入CPU标志位的低8位中：SF:ZF:-:AF:-:PF:-:CF<-AH。

FNSTSW将C3/C2/C0位放入AH寄存器的第6，2，0位中。

换句话说，fnstsw ax/sahf指令对是将C3/C2/C0移入CPU标志位ZF,PF,CF中。

现在我们来回顾一下，C3/C2/C0位会被设置成什么。

```
在我们的例子中，如果a比b大，那么C3/C2/C0位会被设为0，0，0
如果a比b小，这些位会被设为0，0，1
如果a＝b，这些位会被设为1，0，0
```

换句话说，在 FUCOMPP/FNSTSW/SAHF指令后，我们的CPU标志位的状态如下

```
如果a>b,CPU的标志位会被设为:ZF=0,PF=0,CF=0
如果a<b,CPU的标志位会被设为:ZF=0,PF=0,CF=1
如果a=b,CPU的标志位会被设为:ZF=1,PF=0,CF=0
```

SETNBE指令怎样给AL存储0或1：取决于CPU标志位。几乎是JNBE的计数器，利用设置cc码产生的异常，来给AL写入0或1，但是Jccbut Jcc do actual jump or not.SETNBE存储1只在CF=0并且ZF=0的情况下。如果为假，将会存储0。

cf和ZF都为0只存在于一种情况：a>b

然后one将会被存入AL中，接下来JZ不会被触发，函数将返回_a。在其他的情况下，返回的是_b。

带优化的 GCC 4.4.1

GCC 4.4.1-03优化选项turned开关

```
            public d_max
d_max       proc near
arg_0       = qword ptr 8
arg_8       = qword ptr 10h
            push    ebp
            mov     ebp, esp
            fld     [ebp+arg_0] ; _a
            fld     [ebp+arg_8] ; _b

; stack state now: ST(0) = _b, ST(1) = _a
            fxch    st(1)

; stack state now: ST(0) = _a, ST(1) = _b
            fucom   st(1) ; compare _a and _b

            fnstsw  ax
            sahf
            ja      short loc_8048448
; store ST(0) to ST(0) (idle operation), pop value at top of sta
ck, leave _b at top
            fstp    st
            jmp     short loc_804844A

loc_8048448:
; store _a to ST(0), pop value at top of stack, leave _a at top
            fstp    st(1)
loc_804844A:
            pop     ebp
            retn
d_max       endp
```

几乎相同除了一种情况：JA替代了SAHF。事实上，条件跳转指令（JA, JAE, JBE, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ）检查通过检查CF和ZF标志来知晓两个无符号数字的比较结果。C3/C2/C0位在比较之后被放入这些标志位中然后条件跳转就会起效。JA会生效如果CF和ZF都为0。

因此，这里列出的条件跳转指令可以在FNSTSW/SAHF指令对之后使用。

看上去，FPU C3/C2/C0状态位故意放置在那里，传递给CPU而不需要额外的交换。

ARM

ARM+优化Xcode(LLVM)+ARM模式

```
VMOV          D16, R2, R3 ; b
VMOV          D17, R0, R1 ; a
VCMPE.F64     D17, D16
VMRS          APSR_nzcv, FPSCR
VMOVGT.F64    D16, D17 ; copy b to D16
VMOV          R0, R1, D16
BX            LR
```

一个简单例子。输入值放在D17到D16寄存器中，然后借助VCMPE指令进行比较。就像x86协处理器一样，ARM协处理器拥有自己的标志位寄存器（FPSCR），因为存储协处理器的特殊标志需要存储。

就像x86中一样，在ARM中没有条件跳转指令，在协处理器状态寄存器中检查位，因此这里有VMRS指令，从协处理器状态字复制4位（N,Z,C,V）放入通用状态位（APSR寄存器）

VMOVGT类似MOVGT指令，如果比较时一个操作数比其它的大，指令将会被执行。

如果被执行了，b值将会写入D16，暂时被存储在D17中。

如果没有被执行，a的值将会保留在D16寄存器中。

倒数第二个指令VMOV将会通过R0和R1寄存器对准备D16寄存去中的值来返回。

ARM+优化 Xcode（LLVM）+thumb-2 模式

```
VMOV          D16, R2, R3 ; b
VMOV          D17, R0, R1 ; a
VCMPE.F64     D17, D16
VMRS          APSR_nzcv, FPSCR
IT GT
VMOVGT.F64    D16, D17
VMOV          R0, R1, D16
BX            LR
```

几乎和前一个例子一样，有一些小小的不同。事实上，许多ARM中的指令在ARM模式下根据条件判定，当条件为真则执行。

但是在thumb代码中没有这样的事。在16位的指令中没有空闲的4位来编码条件。

但是，thumb-2为老的thumb指令进行扩展使得特殊判断成为可能。

这里是IDA-生成的表单，我们可以看到VMOVGT指令，和在前一个例子中是相同的。

但事实上，常见的VMOV就这样编码，但是IDA加上了—GT后缀，因为以前会放置“IT GT”指令。

IT指令定义所谓的if-then块。指令后面最多放置四条指令是可能的，判断后缀会被加上。在我们的例子中，“IT GT”意味着下一条指令会被执行，如果GT（Greater Than）条件为眞。

下面是一段更加复杂的代码，来源于"愤怒的小鸟"(ios版)

```
ITE NE
VMOVNE    R2, R3, D16
VMOVEQ    R2, R3, D17
```

ITE意味着if-the-else并且它为接下来的两条指令加上后缀。第一条指令将会执行如果ITE（NE,不相等）这时为眞，为假则执行第二条指令。（与NE对立的就是EQ（equal））

这段代码也来自"愤怒的小鸟":

```
ITTTT EQ
MOVEQ       R0, R4
ADDEQ       SP, SP, #0x20
POPEQ.W     {R8,R10}
POPEQ       {R4-R7,PC}
```

4个“T”符号在助记符中意味着接下来的4条指令将会被执行如果条件为眞。这也是IDA在每条指令后面加上-EQ后缀的原因。

如果出现上面例子中ITEEE EQ（if-then-else-else-else）,那么这些后缀将会被这样设置。

```
-EQ
-NE
-NE
-NE
```

另一段来自“愤怒的小鸟”的代码。

```
CMP.W       R0, #0xFFFFFFFF
ITTE LE
SUBLE.W     R10, R0, #1
NEGLE       R0, R0
MOVGT       R10, R0
```

ITTE（if-then-then-else）意味着第一条第二条指令将会被执行，如果LE（Less or Equal）条件为眞，反之第三条指令将会执行。

编译器通常不生成所有的组合。举个例子，在“愤怒的小鸟”中提到的（ios经典版）只有这些IT指令会被使用：IT,ITE,ITT,ITTE,ITTT,ITTTT.我们怎样去学习它呢？在IDA中，产生这些列举的文件是可能的，于是我这么做了，并且设置选项以4字节的格式现实操作码。因为IT操作码的高16位是0xBF，使用grep指令

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

另外，对于thumb-2模式 ARM汇编语言的程序，通过附加的条件后缀，必要的时候汇编会自动加上IT指令和相应的标志。

ARM+非优化模式 Xcode(LLVM)+ARM模式

```
b                =-0x20
a                =-0x18
val_to_return    = -0x10
saved_R7         = -4
                 STR          R7, [SP,#saved_R7]!
                 MOV          R7, SP
                 SUB          SP, SP, #0x1C
                 BIC          SP, SP, #7
                 VMOV         D16, R2, R3
                 VMOV         D17, R0, R1
                 VSTR         D17, [SP,#0x20+a]
                 VSTR         D16, [SP,#0x20+b]
                 VLDR         D16, [SP,#0x20+a]
                 VLDR         D17, [SP,#0x20+b]
                 VCMPE.F64    D16, D17
                 VMRS         APSR_nzcv, FPSCR
                 BLE          loc_2E08
                 VLDR         D16, [SP,#0x20+a]
                 VSTR         D16, [SP,#0x20+val_to_return]
                 B            loc_2E10
loc_2E08
                 VLDR         D16, [SP,#0x20+b]
                 VSTR         D16, [SP,#0x20+val_to_return]
loc_2E10
                 VLDR         D16, [SP,#0x20+val_to_return]
                 VMOV         R0, R1, D16
                 MOV          SP, R7
                 LDR          R7, [SP+0x20+b],#4
                 BX           LR
```

基本和我们看到的一样，但是太多冗陈代码，因为a和b的变量存储在本地栈中，还有返回値

ARM+优化模式keil＋thumb模式

```
        PUSH    {R3-R7,LR}
        MOVS    R4, R2
        MOVS    R5, R3
        MOVS    R6, R0
        MOVS    R7, R1
        BL      __aeabi_cdrcmple
        BCS     loc_1C0
        MOVS    R0, R6
        MOVS    R1, R7
        POP     {R3-R7,PC}
loc_1C0
        MOVS    R0, R4
        MOVS    R1, R5
        POP     {R3-R7,PC}
```

keil 不为浮点数的比较生成特殊的指令，因为他不能依靠核心CPU的支持，它也不能直接按位比较。这里有一个外部函数用于比较：__aeabi_cdrcmple. N.B. 比较的结果用来设置标志，因此接下来的BCS（标志位设置 - 大于或等于）指令可能有效并且无需额外的代码。

17.7.3 ARM64

Optimizing GCC (Linaro) 4.9

Non-optimizing GCC (Linaro) 4.9

Exercise

Optimizing GCC (Linaro) 4.9—float

17.7.4 MIPS

17.8 栈,计算器和逆波兰表示法

17.9 x64

17.10 练习

第十八章

数组

数组是在内存中连续排列的一组变量，这些变量具有相同类型1。

18.1 小例子

```
#include <stdio.h>
int main()
{
    int a[20];
    int i;
    for (i=0; i<20; i++)
        a[i]=i*2;
    for (i=0; i<20; i++)
        printf ("a[%d]=%d", i, a[i]);
    return 0;
};
```

18.1.1 x86

MSVC

编译后：

Listing 18.1: MSVC

```
_TEXT   SEGMENT
_i$ = -84                                          ; size = 4
_a$ = -80                                          ; size = 80
_main        PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84          ; 00000054H
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN6@main
$LN5@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp     DWORD PTR _i$[ebp], 20 ; 00000014H
    jge     SHORT $LN4@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN5@main
$LN4@main:
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 20 ; 00000014H
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    mov     edx, DWORD PTR _a$[ebp+ecx*4]
    push    edx
    mov     eax, DWORD PTR _i$[ebp]
    push    eax
    push    OFFSET $SG2463
    call    _printf
    add     esp, 12 ; 0000000cH
    jmp     SHORT $LN2@main
$LN1@main:
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
```

这段代码主要有两个循环：第一个循环填充数组，第二个循环打印数组元素。shl ecx,1指令使ecx的值乘以2，更多关于左移请参考17.3.1。 在堆栈上为数组分配了80个字节的空间，包含20个元素，每个元素4字节大小。

Let’s try this example in OllyDbg 缺漏 =

GCC

GCC 4.4.1编译后为：

Listing 18.2: GCC 4.4.1

```
                public main
main            proc near                       ; DATA XREF: _start+
17

var_70          = dword ptr -70h
var_6C          = dword ptr -6Ch
var_68          = dword ptr -68h
i_2             = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 70h
                mov     [esp+70h+i], 0 ; i=0
                jmp     short loc_804840A
loc_80483F7:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+70h+i]
                add     edx, edx ; edx=i*2
                mov     [esp+eax*4+70h+i_2], edx
                add     [esp+70h+i], 1 ; i++
loc_804840A:
                cmp     [esp+70h+i], 13h
                jle     short loc_80483F7
                mov     [esp+70h+i], 0
                jmp     short loc_8048441
loc_804841B:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+eax*4+70h+i_2]
                mov     eax, offset aADD ; "a[%d]=%d
"
                mov     [esp+70h+var_68], edx
                mov     edx, [esp+70h+i]
                mov     [esp+70h+var_6C], edx
                mov     [esp+70h+var_70], eax
                call    _printf
                add     [esp+70h+i], 1
loc_8048441:
                cmp     [esp+70h+i], 13h
                jle     short loc_804841B
                mov     eax, 0
                leave
                retn
main            endp
```

顺便提一下，一个int类型（指向*int*的指针）的变量—你可以使该变量指向数组并将该数组传递给另一个函数，更准确的说，传递的指针指向数组的第一个元素（该数组其它元素的地址需要显示计算）。比如*a[idx]*，*idx*加上指向该数组的指针并返回该元素。 一个有趣的例子：类似*"string"*字符数组的类型是*const char*，索引可以应用与该指针。比如可能写作"string"[i]—正确的C/C++表达式。

18.1.2 ARM

ARM + Non-optimizing Keil + ARM mode

```
        EXPORT _main
_main
        STMFD   SP!, {R4,LR}
        SUB     SP, SP, #0x50           ; allocate place for 20
int variables
; first loop
        MOV     R4, #0                  ; i
        B       loc_4A0
loc_494
        MOV     R0, R4,LSL#1            ; R0=R4*2
        STR     R0, [SP,R4,LSL#2]       ; store R0 to SP+R4<<2 (
same as SP+R4*4)
        ADD     R4, R4, #1              ; i=i+1
loc_4A0
        CMP     R4, #20                 ; i<20?
        BLT     loc_494                 ; yes, run loop body aga
in
; second loop
        MOV     R4, #0                  ; i
        B       loc_4C4
loc_4B0
        LDR     R2, [SP,R4,LSL#2]       ; (second printf argumen
t) R2=*(SP+R4<<4) (same as *(SP+R4*4))
        MOV     R1, R4                  ; (first printf argument
) R1=i
        ADR     R0, aADD                ; "a[%d]=%d
"
        BL      __2printf
        ADD     R4, R4, #1              ; i=i+1
loc_4C4
        CMP     R4, #20                 ; i<20?
        BLT     loc_4B0                 ; yes, run loop body aga
in
        MOV     R0, #0                  ; value to return
        ADD     SP, SP, #0x50           ; deallocate place, allo
cated for 20 int variables
        LDMFD   SP!, {R4,PC}
```

int类型长度为32bits即4字节，20个int变量需要80（0x50）字节，因此“sub sp,sp,#0x50”指令为在栈上分配存储空间。 两个循环迭代器i被存储在R4寄存器中。 值i2被写入数组，通过将*i*值左移*1*位实现乘以*2*的效果，整个过程通过*”MOV R0,R4,LSL#1*指令来实现。 *“STR R0, [SP,R4,LSL#2]”*把*R0*内容写入数组。过程为：*SP*指向数组开始，*R4*是*i*，*i*左移*2*位相当于乘以*4*，即(SP+R4*4)=R0*。 第二个*loop*的*“LDR R2, [SP,R4,LSL#2]”*从数组读取数值到寄存器，*R2*=(SP+R4*4)。

ARM + Keil + thumb 模式优化后

```
_main
        PUSH    {R4,R5,LR}
; allocate place for 20 int variables + one more variable
        SUB     SP, SP, #0x54
; first loop
        MOVS    R0, #0                  ; i
        MOV     R5, SP                  ; pointer to first array
 element
loc_1CE
        LSLS    R1, R0, #1              ; R1=i<<1 (same as i*2)
        LSLS    R2, R0, #2              ; R2=i<<2 (same as i*4)
        ADDS    R0, R0, #1              ; i=i+1
        CMP     R0, #20                 ; i<20?
        STR     R1, [R5,R2]             ; store R1 to *(R5+R2) (
same R5+i*4)
        BLT     loc_1CE                 ; yes, i<20, run loop bo
dy again
; second loop
        MOVS    R4, #0                  ; i=0
loc_1DC
        LSLS    R0, R4, #2              ; R0=i<<2 (same as i*4)
        LDR     R2, [R5,R0]             ; load from *(R5+R0) (sa
me as R5+i*4)
        MOVS    R1, R4
        ADR     R0, aADD                ; "a[%d]=%d
"
        BL      __2printf
        ADDS    R4, R4, #1              ; i=i+1
        CMP     R4, #20                 ; i<20?
        BLT     loc_1DC                 ; yes, i<20, run loop bo
dy again
        MOVS    R0, #0                  ; value to return
; deallocate place, allocated for 20 int variables + one more va
riable
        ADD     SP, SP, #0x54
        POP     {R4,R5,PC}
```

Thumb代码也是非常类似的。Thumb模式计算数组偏移的移位操作使用特定的指令LSLS。 编译器在堆栈中申请的数组空间更大，但是最后4个字节的空间未使用。

Non-optimizing GCC 4.9.1 (ARM64)

18.1.3 MIPS

18.2 缓冲区溢出

18.2.1 读取外部数组的边界

Array[index]中index指代数组索引，仔细观察下面的代码，你可能注意到代码没有index是否小于20。如果index大于20？这是C/C++经常被批评的特征。 以下代码可以成功编译可以工作：

```
#include <stdio.h>
int main()
{
    int a[20];
    int i;
    for (i=0; i<20; i++)
        a[i]=i*2;
    printf ("a[100]=%d", a[100]);
    return 0;
};
```

编译后 (MSVC 2010)：

```
_TEXT   SEGMENT
_i$ = -84                               ; size = 4
_a$ = -80                               ; size = 80
_main       PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84                 ; 00000054H
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 20  ; 00000014H
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN2@main
$LN1@main:
    mov     eax, DWORD PTR _a$[ebp+400]
    push    eax
    push    OFFSET $SG2460
    call    _printf
    add     esp, 8
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main       ENDP
```

运行，我们得到： a[100]=760826203

打印的数字仅仅是距离数组第一个元素400个字节处的堆栈上的数值。 编译器可能会自动添加一些判断数组边界的检测代码（更高级语言3），但是这可能影响运行速度。 我们可以从栈上非法读取数值，是否可以写入数值呢？ 下面我们将写入数值：

```
#include <stdio.h>
int main()
{
    int a[20];
    int i;

    for (i=0; i<30; i++)
        a[i]=i;

    return 0;
};
```

我们得到：

```
_TEXT   SEGMENT
_i$ = -84                                  ; size = 4
_a$ = -80                                  ; size = 80
_main       PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84 ; 00000054H
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 30 ; 0000001eH
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    mov     edx, DWORD PTR _i$[ebp] ; that instruction is obviou
sly redundant
    mov     DWORD PTR _a$[ebp+ecx*4], edx ; ECX could be used as
 second operand here instead
    jmp     SHORT $LN2@main
$LN1@main:
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main       ENDP
```

编译后运行，程序崩溃。我们找出导致崩溃的地方。 没有使用调试器，而是使用我自己写的小工具tracer足以完成任务。 我们用它看被调试进程崩溃的地方：

```
generic tracer 0.4 (WIN32), http://conus.info/gt

New process: C:PRJ...1.exe, PID=7988
EXCEPTION_ACCESS_VIOLATION: 0x15 (<symbol (0x15) is in unknown m
odule>), ExceptionInformation
[0]=8
EAX=0x00000000 EBX=0x7EFDE000 ECX=0x0000001D EDX=0x0000001D
ESI=0x00000000 EDI=0x00000000 EBP=0x00000014 ESP=0x0018FF48
EIP=0x00000015
FLAGS=PF ZF IF RF
PID=7988|Process exit, return code -1073740791
```

我们来看各个寄存器的状态，异常发生在地址0x15。这是个非法地址—至少对win32代码来说是！这种情况并不是我们期望的，我们还可以看到EBP值为0x14,ECX和EDX都为0x1D。 让我们来研究堆栈布局。 代码进入main（）后，EBP寄存器的值被保存在栈上。为数组和变量i一共分配84字节的栈空间，即(20+1)*sizeof(int)。此时ESP指向_i变量，之后执行push something，something将紧挨着_i。 此时main()函数内栈布局为：

```
ESP
ESP+4
ESP+84
ESP+88
4 bytes for i
80 bytes for a[20] array
saved EBP value
returning address
```

指令a[19]=something写入最后的int到数组边界（这里是数组边界！）。 指令a[20]=something，something将覆盖栈上保存的EBP值。 请注意崩溃时寄存器的状态。在此例中，数字20被写入第20个元素，即原来存放EBP值得地方被写入了20（20的16进制表示是0x14）。然后RET指令被执行，相当于执行POP EIP指令。 RET指令从堆栈中取出返回地址（该地址为CRT内部调用main()的地址），返回地址处被存储了21（0x15）。CPU执行地址0x15的代码，异常被抛出。Welcome！这被称为缓冲区溢出4。 使用字符数组代替int数组，创建一个较长的字符串，把字符串传递给程序，函数没有检测字符串长度，把字符复制到较短的缓冲区，你能够找到找到程序必须跳转的地址。事实上，找出它们并不是很简单。 我们来看GCC 4.4.1编译后的同类代码：

```
            public main
main        proc near

a           = dword ptr -54h
i           = dword ptr -4

            push    ebp
            mov     ebp, esp
            sub     esp, 60h
            mov     [ebp+i], 0
            jmp     short loc_80483D1
loc_80483C3:
            mov     eax, [ebp+i]
            mov     edx, [ebp+i]
            mov     [ebp+eax*4+a], edx
            add     [ebp+i], 1
loc_80483D1:
            cmp     [ebp+i], 1Dh
            jle     short loc_80483C3
            mov     eax, 0
            leave
            retn
main        endp
```

在linux下运行将产生：段错误。使用GDB调试：

```
(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax         0x0             0
ecx         0xd2f96388      -755407992
edx         0x1d            29
ebx         0x26eff4        2551796
esp         0xbffff4b0      0xbffff4b0
ebp         0x15            0x15
esi         0x0             0
edi         0x0             0
eip         0x16            0x16
eflags      0x10202         [ IF RF ]
cs          0x73            115
ss          0x7b            123
ds          0x7b            123
es          0x7b            123
fs          0x0             0
gs          0x33            51
(gdb)
```

寄存器的值与win32例子略微不同，因为堆栈布局也不太一样。

18.2.2 Writing beyond array bounds

MSVC

GCC

18.3 防止缓冲区溢出的方法

下面一些方法防止缓冲区溢出。MSVC使用以下编译选项：

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

一种方法是在函数局部变量和序言之间写入随机值。在函数退出之前检查该值。如果该值不一致则挂起而不执行RET。进程将被挂起。 该随机值有时被称为“探测值”。 如果使用MSVC编译简单的例子（18.1），使用RTC1和RTCs选项，将能看到函数调用@_RTC_CheckStackVars@8函数来检测“探测值“。

我们来看GCC如何处理这些。我们使用alloca()(4.2.4)例子：

```
#include <malloc.h>
#include <stdio.h>
void f()
{
    char *buf=(char*)alloca (600);
    _snprintf (buf, 600, "hi! %d, %d, %d", 1, 2, 3);

    puts (buf);
};
```

我们不使用任何附加编译选项，只使用默认选项，GCC 4.7.3将插入“探测“检测代码：

Listing 18.3: GCC 4.7.3

```
.LC0:
    .string "hi! %d, %d, %d
"
f:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 676
    lea     ebx, [esp+39]
    and     ebx, -16
    mov     DWORD PTR [esp+20], 3
    mov     DWORD PTR [esp+16], 2
    mov     DWORD PTR [esp+12], 1
    mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0     ; "hi! %d, %
d, %d
"
    mov     DWORD PTR [esp+4], 600
    mov     DWORD PTR [esp], ebx
    mov     eax, DWORD PTR gs:20                    ; canary
    mov     DWORD PTR [ebp-12], eax
    xor     eax, eax
    call    _snprintf
    mov     DWORD PTR [esp], ebx
    call    puts
    mov     eax, DWORD PTR [ebp-12]
    xor     eax, DWORD PTR gs:20                    ; canary
    jne     .L5
    mov     ebx, DWORD PTR [ebp-4]
    leave
    ret
.L5:
call __stack_chk_fail
```

随机值存在于gs:20。它被写入到堆栈，在函数的结尾与gs:20的探测值对比，如果不一致，__stack_chk_fail函数将被调用，控制台(Ubuntu 13.04 x86)将输出以下信息：

```
*** buffer overflow detected ***: ./2_1 terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5
c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935
]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 2097586 /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586 /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586 /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0 [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602 /lib/i386-linux-gn
u/libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602 /lib/i386-linux-gn
u/libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602 /lib/i386-linux-gn
u/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781 /lib/i386-linux-gn
u/libc-2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781 /lib/i386-linux-gn
u/libc-2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781 /lib/i386-linux-gn
u/libc-2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0 [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794 /lib/i386-linux-gn
u/ld-2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794 /lib/i386-linux-gn
u/ld-2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794 /lib/i386-linux-gn
u/ld-2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0 [stack]
Aborted (core dumped)
```

gs被叫做段寄存器，这些寄存器被广泛用在MS-DOS和扩展DOS时代。现在的作用和以前不同。简要的说，gs寄存器在linux下一直指向TLS（48）--存储线程的各种信息（win32环境下，fs寄存器同样的作用，指向TIB8 9）。 更多信息请参考linux源码arch/x86/include/asm/stackprotector.h（至少3.11版本）。

18.3.1 Optimizing Xcode (LLVM) + thumb-2 mode

我们回头看简单的数组例子(18.1)。我们来看LLVM如何检查“探测值“。

```
_main
var_64      = -0x64
var_60      = -0x60
var_5C      = -0x5C
var_58      = -0x58
var_54      = -0x54
var_50      = -0x50
var_4C      = -0x4C
var_48      = -0x48
var_44      = -0x44
var_40      = -0x40
var_3C      = -0x3C
var_38      = -0x38
var_34      = -0x34
var_30      = -0x30
var_2C      = -0x2C
var_28      = -0x28
var_24      = -0x24
var_20      = -0x20
var_1C      = -0x1C
var_18      = -0x18
canary      = -0x14
var_10      = -0x10

            PUSH    {R4-R7,LR}
            ADD     R7, SP, #0xC
            STR.W   R8, [SP,#0xC+var_10]!
            SUB     SP, SP, #0x54
            MOVW    R0, #aObjc_methtype ; "objc_methtype"
            MOVS    R2, #0
            MOVT.W  R0, #0
            MOVS    R5, #0
            ADD     R0, PC
            LDR.W   R8, [R0]
            LDR.W   R0, [R8]
            STR     R0, [SP,#0x64+canary]
            MOVS    R0, #2
            STR     R2, [SP,#0x64+var_64]
            STR     R0, [SP,#0x64+var_60]
            MOVS    R0, #4
            STR     R0, [SP,#0x64+var_5C]
            MOVS    R0, #6
            STR     R0, [SP,#0x64+var_58]
            MOVS    R0, #8
            STR     R0, [SP,#0x64+var_54]
            MOVS    R0, #0xA
            STR     R0, [SP,#0x64+var_50]
            MOVS    R0, #0xC
            STR     R0, [SP,#0x64+var_4C]
            MOVS    R0, #0xE
```

```
            STR     R0, [SP,#0x64+var_48]
            MOVS    R0, #0x10
            STR     R0, [SP,#0x64+var_44]
            MOVS    R0, #0x12
            STR     R0, [SP,#0x64+var_40]
            MOVS    R0, #0x14
            STR     R0, [SP,#0x64+var_3C]
            MOVS    R0, #0x16
            STR     R0, [SP,#0x64+var_38]
            MOVS    R0, #0x18
            STR     R0, [SP,#0x64+var_34]
            MOVS    R0, #0x1A
            STR     R0, [SP,#0x64+var_30]
            MOVS    R0, #0x1C
            STR     R0, [SP,#0x64+var_2C]
            MOVS    R0, #0x1E
            STR     R0, [SP,#0x64+var_28]
            MOVS    R0, #0x20
            STR     R0, [SP,#0x64+var_24]
            MOVS    R0, #0x22
            STR     R0, [SP,#0x64+var_20]
            MOVS    R0, #0x24
            STR     R0, [SP,#0x64+var_1C]
            MOVS    R0, #0x26
            STR     R0, [SP,#0x64+var_18]
            MOV     R4, 0xFDA ; "a[%d]=%d
"
            MOV     R0, SP
            ADDS    R6, R0, #4
            ADD     R4, PC
            B loc_2F1C
; second loop begin

loc_2F14
            ADDS    R0, R5, #1
            LDR.W   R2, [R6,R5,LSL#2]
            MOV     R5, R0
loc_2F1C
            MOV     R0, R4
            MOV     R1, R5
            BLX     _printf
            CMP     R5, #0x13
            BNE     loc_2F14
            LDR.W   R0, [R8]
            LDR     R1, [SP,#0x64+canary]
            CMP     R0, R1
            ITTTT   EQ              ; canary still correct?
            MOVEQ   R0, #0
            ADDEQ   SP, SP, #0x54
            LDREQ.W R8, [SP+0x64+var_64],#4
            POPEQ   {R4-R7,PC}
            BLX     ___stack_chk_fail
```

首先可以看到，LLVM循环展开写入数组，LLVM认为先计算出数组元素的值速度更快。 在函数的结尾我们能看到“探测值“的检测—局部存储的值与R8指向的标准值对比。如果相等4指令块将通过”ITTTT EQ“触发，R0写入0，函数退出。如果不相等，指令块将不会被触发，跳向___stack_chk_fail函数，结束进程。

18.4 One more word about arrays

现在我们来理解下面的C/C++代码为什么不能正常使用10：

```
void f(int size)
{
    int a[size];
    ...
};
```

这是因为在编译阶段编译器不知道数组的具体大小无论是在堆栈或者数据段，无法分配具体空间。 如果你需要任意大小的数组，应该通过malloc()分配空间，然后访问内存块来访问你需要的类型数组。或者使用C99标准[15,6.7.5/2]，但它内部看起来更像alloca()(4.2.4)。

18.5 指向字符串的数组

18.5.1 x64

32-bit MSVC

18.5.2 32-bit ARM

ARM in ARM mode

ARM in Thumb mode

18.5.3 ARM64

18.5.4 MIPS

18.5.5 数组溢出

数组溢出保护

18.6 多维数组

多维数组和线性数组在本质上是一样的。 因为计算机内存是线性的，它是一维数组。但是一维数组可以很容易用来表现多维的。 比如数组a[3][4]元素可以放置在一维数组的12个单元中：

```
[0][0]
[0][1]
[0][2]
[0][3]
[1][0]
[1][4]
[1][5]
[1][6]
[2][0]
[2][7]
[2][8]
[2][9]
```

该二维数组在内存中用一维数组索引表示为：

	1	2	3
4	5	6	7
8	9	10	11

为了计算我们需要的元素地址，首先，第一个索引乘以4（矩阵宽度），然后加上第二个索引。这种被称为行优先，C/C++和Python常用这种方法。行优先的意思是：先写入第一行，接着是第二行，...，最后是最后一行。 另一种方法就是列优先，主要用在FORTRAN,MATLAB,R等。列优先的意思是：先写入第一列，然后是第二列，...，最后是最后一列。

18.6.1 二维数组的例子

行填充的例子

列填充的例子

18.6.2 像一位数组那样访问二维数组

18.6.3 多维数组

多维数组与此类似。 我们看个例子：

Listing 18.4: simple example

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
    a[x][y][z]=value;
};
```

x86

MSVC 2010：

Listing 18.5: MSVC 2010

```
_DATA   SEGMENT
COMM    _a:DWORD:01770H
_DATA   ENDS
PUBLIC  _insert
_TEXT   SEGMENT
_x$ = 8         ; size = 4
_y$ = 12        ; size = 4
_z$ = 16        ; size = 4
_value$ = 20    ; size = 4
_insert     PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _x$[ebp]
    imul    eax, 2400                   ; eax=600*4*x
    mov     ecx, DWORD PTR _y$[ebp]
    imul    ecx, 120                    ; ecx=30*4*y
    lea     edx, DWORD PTR _a[eax+ecx]  ; edx=a + 600*4*x + 30*4
*y
    mov     eax, DWORD PTR _z$[ebp]
    mov     ecx, DWORD PTR _value$[ebp]
    mov     DWORD PTR [edx+eax*4], ecx  ; *(edx+z*4)=value
    pop     ebp
    ret     0
_insert ENDP
_TEXT ENDS
```

多维数组计算索引公式：address=600*4*x+30*4*y+4z。因为int类型为32-bits（4字节），所以要乘以4。

Listing 18.6: GCC 4.4.1

```
        public insert
insert  proc near
x       = dword ptr 8
y       = dword ptr 0Ch
z       = dword ptr 10h
value   = dword ptr 14h
        push    ebp
        mov     ebp, esp
        push    ebx
        mov     ebx, [ebp+x]
        mov     eax, [ebp+y]
        mov     ecx, [ebp+z]
        lea     edx, [eax+eax] ; edx=y*2
        mov     eax, edx ; eax=y*2
        shl     eax, 4 ; eax=(y*2)<<4 = y*2*16 = y*32
        sub     eax, edx ; eax=y*32 - y*2=y*30
        imul    edx, ebx, 600 ; edx=x*600
        add     eax, edx ; eax=eax+edx=y*30 + x*600
        lea     edx, [eax+ecx] ; edx=y*30 + x*600 + z
        mov     eax, [ebp+value]
        mov     dword ptr ds:a[edx*4], eax ; *(a+edx*4)=value
        pop     ebx
        pop     ebp
        retn
insert  endp
```

GCC使用的不同的计算方法。为计算第一个操作值30y，GCC没有使用乘法指令。GCC的做法是：(???? + ????) ≪ 4 − (???? + ????) = (2????) ≪ 4 − 2???? = 2 · 16 · ???? − 2???? = 32???? − 2???? = 30????。因此30y的计算仅使用加法和移位操作，这样速度更快。

ARM + Non-optimizing Xcode (LLVM) + thumb mode

Listing 18.7: Non-optimizing Xcode (LLVM) + thumb mode

```
_insert

value          = -0x10
z              = -0xC
y              = -8
x              = -4

; allocate place in local stack for 4 values of int type
SUB            SP, SP, #0x10
MOV            R9, 0xFC2 ; a
ADD            R9, PC
LDR.W          R9, [R9]
STR            R0, [SP,#0x10+x]
STR            R1, [SP,#0x10+y]
STR            R2, [SP,#0x10+z]
STR            R3, [SP,#0x10+value]
LDR            R0, [SP,#0x10+value]
LDR            R1, [SP,#0x10+z]
LDR            R2, [SP,#0x10+y]
LDR            R3, [SP,#0x10+x]
MOV            R12, 2400
MUL.W          R3, R3, R12
ADD            R3, R9
MOV            R9, 120
MUL.W          R2, R2, R9
ADD            R2, R3
LSLS           R1, R1, #2 ; R1=R1<<2
ADD            R1, R2
STR            R0, [R1] ; R1 - address of array element
; deallocate place in local stack, allocated for 4 values of int
 type
ADD            SP, SP, #0x10
BX             LR
```

非优化的LLVM代码在栈中保存了所有变量，这是冗余的。元素地址的计算我们通过公式已经找到了。

ARM + Optimizing Xcode (LLVM) + thumb mode

Listing 18.8: Optimizing Xcode (LLVM) + thumb mode

```
_insert
MOVW        R9, #0x10FC
MOV.W       R12, #2400
MOVT.W      R9, #0
RSB.W       R1, R1, R1,LSL#4    ; R1 - y. R1=y<<4 - y = y*16 - y
 = y*15
ADD         R9, PC ; R9 = pointer to a array
LDR.W       R9, [R9]
MLA.W       R0, R0, R12, R9     ; R0 - x, R12 - 2400, R9 - point
er to a. R0=x*2400 + ptr to a
ADD.W       R0, R0, R1,LSL#3    ; R0 = R0+R1<<3 = R0+R1*8 = x*24
00 + ptr to a + y*15*8 =
                                ; ptr to a + y*30*4 + x*600*4
STR.W       R3, [R0,R2,LSL#2]   ; R2 - z, R3 - value. address=R0
+z*4 =
                                ; ptr to a + y*30*4 + x*600*4 +
z*4
BX          LR
```

这里的小技巧没有使用乘法，使用移位、加减法等。 这里有个新指令RSB（逆向减法），该指令的意义是让第一个操作数像SUB第二个操作数一样可以应用LSL#4附加操作。 “LDR.W R9, [R9]”类似于x86下的LEA指令（B.6.2），这里什么都没有做，是冗余的。显然，编译器没有优化它。

MIPS

18.6.4 更多的例子

18.7 讲打包的字符串作为数组

18.7.1 32-bit ARM

18.7.2 ARM64

18.7.3 MIPS

18.7.4 Conclusion

18.8 结论

18.9 练习

第十九章

操纵特殊的bit

很多函数参数的输入标志使用了位域。当然，可以使用bool类型来替代，只是有点浪费。

19.1 Specific bit checking

x86

Win32 API 例子:

```
    HANDLE fh;

    fh=CreateFile("file", GENERIC_WRITE | GENERIC_READ, FILE_SHA
RE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

MSVC 2010： Listing 17.1: MSVC 2010

```
push    0
push    128             ; 00000080H
push    4
push    0
push    1
push    -1073741824     ; c0000000H
push    OFFSET $SG78813
call    DWORD PTR __imp__CreateFileA@28
mov     DWORD PTR _fh$[ebp], eax
```

我们再查看WinNT.h:

Listing 17.2: WinNT.h

```
#define GENERIC_READ (0x80000000L)
#define GENERIC_WRITE (0x40000000L)
#define GENERIC_EXECUTE (0x20000000L)
#define GENERIC_ALL (0x10000000L)
```

容易看出GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000，该值作为CreateFile()1函数的第二个参数。 CreateFile()如何检查该标志呢？ 以Windows XP SP3 x86为例，在kernel32.dll中查看CreateFileW检查该标志的代码片段： Listing 17.3: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429 test byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D mov [ebp+var_8], 1
.text:7C83D434 jz short loc_7C83D417
.text:7C83D436 jmp loc_7C810817
```

我们来看TEST指令，该指令并未检测整个第二个参数，仅检测关键的一个字节(ebp+dwDesiredAccess+3)，检测0x40标志（这里代表GENERIC_WRITE标志）。 Test对两个参数(目标，源)执行AND逻辑操作,并根据结果设置标志寄存器,结果本身不会保存（CMP和SUB与此类似（6.6.1））。 该代码片段逻辑如下：

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

如果AND指令没有设置ZF位，JZ将不触发跳转。如果dwDesiredAccess不等于0x40000000，AND结果将是0，ZF位将会被设置，条件跳转将被触发。

我们在linux GCC 4.4.1下查看：

```
#include <stdio.h>
#include <fcntl.h>
void main()
{
    int handle;

    handle=open ("file", O_RDWR | O_CREAT);
};
```

我们得到： Listing 17.4: GCC 4.4.1

```
    public main
main proc near


var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_4 = dword ptr -4


    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    sub esp, 20h
    mov [esp+20h+var_1C], 42h
    mov [esp+20h+var_20], offset aFile ; "file"
    call _open
    mov [esp+20h+var_4], eax
    leave
    retn
main endp
```

我们在libc.so.6库中查看open()函数，看到syscall： Listing 17.5: open() (libc.so.6)

```
.text:000BE69B mov edx, [esp+4+mode] ; mode
.text:000BE69F mov ecx, [esp+4+flags] ; flags
.text:000BE6A3 mov ebx, [esp+4+filename] ; filename
.text:000BE6A7 mov eax, 5
.text:000BE6AC int 80h ; LINUX - sys_open
```

因此open()对于标志位的检测在内核中。 对于linux2.6，当sys_open被调用时，最终传递到do_sys_open内核函数，然后进入do_filp_open()函数（该函数位于源码fs/namei.c中）。 除了通过堆栈传递参数，还可以通过寄存器传递方式，这种调用方式成为fastcall(47.3)。这种调用方式CPU不需要访问堆栈就可以直接读取参数的值，所以速度更快。GCC有编译选项regram2，可以设置通过寄存器传递的参数的个数。 Linux2.6内核编译附加选项为-mregram=33 4。 这意味着前3个参数通过EAX、EDX、ECX寄存器传递，剩余的参数通过堆栈传递。如果参数小于3，仅部分寄存器被使用。 我们下载linux内核2.6.31源码，在Ubuntu中编译：make vmlinux，在IDA中打开，找到do_filp_open()函数。在开始部分我们可以看到（注释个人添加）： Listing 17.6:do_filp_open() (linux kernel 2.6.31)

```
do_filp_open proc near
...
    push ebp
    mov ebp, esp
    push edi
    push esi
    push ebx
    mov ebx, ecx
    add ebx, 1
    sub esp, 98h
    mov esi, [ebp+arg_4] ; acc_mode (5th arg)
    test bl, 3
    mov [ebp+var_80], eax ; dfd (1th arg)
    mov [ebp+var_7C], edx ; pathname (2th arg)
    mov [ebp+var_78], ecx ; open_flag (3th arg)
    jnz short loc_C01EF684
    mov ebx, ecx ; ebx <- open_flag
```

GCC保存3个参数的值到堆栈。否则，可能会造成寄存器浪费。 我们来看代码片段： Listing 17.7: do_filp_open() (linux kernel 2.6.31)

```
loc_C01EF6B4:             ; CODE XREF: do_filp_open+4F
    test bl, 40h          ; O_CREAT
    jnz loc_C01EF810
    mov edi, ebx
    shr edi, 11h
    xor edi, 1
    and edi, 1
    test ebx, 10000h
    jz short loc_C01EF6D3
    or edi, 2
```

O_CREAT宏等于0x40，如果open_flag为0x40，标志位被置1，接下来的JNZ指令将被触发。

ARM

Linux kernel3.8.0检测O_CREAT过程有点不同。 Listing 17.8: linux kernel 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname, co
nst struct open_flags *op)
{
... filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RC
U); ...
}

static struct file *path_openat(int dfd, struct filename *pathna
me, struct nameidata *nd, const struct open_flags *op, int flags
)

{
... error = do_last(nd, &path, file, op, &opened, pathname); ...

}

static int do_last(struct nameidata *nd, struct path *path, stru
ct file *file, const struct open_flags *op, int *opened, struct
filename *name)
{
    ...
    if (!(open_flag & O_CREAT)) {
        ...
        error = lookup_fast(nd, path, &inode);
        ...
    } else {
        ... error = complete_walk(nd);
    }
    ...
}
```

在IDA中查看ARM模式内核： Listing 17.9: do_last() (vmlinux)

```
...
.text:C0169EA8 MOV           R9, R3 ; R3 - (4th argument) open_f
lag
...
.text:C0169ED4 LDR           R6, [R9] ; R6 - open_flag
...
.text:C0169F68 TST           R6, #0x40 ; jumptable C0169F00 defa
ult case
.text:C0169F6C BNE           loc_C016A128
.text:C0169F70 LDR           R2, [R4,#0x10]
.text:C0169F74 ADD           R12, R4, #8
.text:C0169F78 LDR           R3, [R4,#0xC]
.text:C0169F7C MOV           R0, R4
.text:C0169F80 STR           R12, [R11,#var_50]
.text:C0169F84 LDRB          R3, [R2,R3]
.text:C0169F88 MOV           R2, R8
.text:C0169F8C CMP           R3, #0
.text:C0169F90 ORRNE         R1, R1, #3
.text:C0169F94 STRNE         R1, [R4,#0x24]
.text:C0169F98 ANDS          R3, R6, #0x200000
.text:C0169F9C MOV           R1, R12
.text:C0169FA0 LDRNE         R3, [R4,#0x24]
.text:C0169FA4 ANDNE         R3, R3, #1
.text:C0169FA8 EORNE         R3, R3, #1
.text:C0169FAC STR           R3, [R11,#var_54]
.text:C0169FB0 SUB           R3, R11, #-var_38
.text:C0169FB4 BL            lookup_fast
...
.text:C016A128 loc_C016A128          ; CODE XREF: do_last.isra.1
4+DC
.text:C016A128 MOV           R0, R4
.text:C016A12C BL            complete_walk
...
```

TST指令类似于x86下的TEST指令。 这段代码来自do_last()函数源码，有两个分支lookup_fast()和complete_walk()。这里O_CREAT宏也等于0x40。

19.2 Specific bit setting/clearing

例如：

```
#define IS_SET(flag, bit) ((flag) & (bit))
#define SET_BIT(var, bit) ((var) |= (bit))
#define REMOVE_BIT(var, bit) ((var) &= ~(bit))
int f(int a)
{
    int rt=a;
    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};
```

19.2.1 x86

Non-optimizing MSVC

MSVC 2010: Listing 17.10: MSVC 2010

```
_rt$ = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
push ebp
mov ebp, esp
push ecx
mov eax, DWORD PTR _a$[ebp]
mov DWORD PTR _rt$[ebp], eax
mov ecx, DWORD PTR _rt$[ebp]
or ecx, 16384            ; 00004000H
mov DWORD PTR _rt$[ebp], ecx
mov edx, DWORD PTR _rt$[ebp]
and edx, -513            ; fffffdffH
mov DWORD PTR _rt$[ebp], edx
mov eax, DWORD PTR _rt$[ebp]
mov esp, ebp
pop ebp
ret 0
_f ENDP
```

OR指令添加一个或多个bit位而忽略了其余位。 AND用来重置一个bit位。

OllyDbg

Optimizing MSVC

如果我们使用msvc编译，并且打开优化选项(/Ox)，代码将会更短： Listing 17.11: Optimizing MSVC

```
_a$ = 8          ; size = 4
_f PROC
    mov eax, DWORD PTR _a$[esp-4]
    and eax, -513    ; fffffdffH
    or eax, 16384    ; 00004000H
    ret 0
_f ENDP
```

Non-optimizing GCC

我们来看GCC 4.4.1无优化的代码：

```
            public f
f           proc near
var_4       = dword ptr -4
arg_0       = dword ptr 8
        push ebp
        mov ebp, esp
        sub esp, 10h
        mov eax, [ebp+arg_0]
        mov [ebp+var_4], eax
        or [ebp+var_4], 4000h
        and [ebp+var_4], 0FFFFFDFFh
        mov eax, [ebp+var_4]
        leave
        retn
f           endp
```

Optimizing GCC

MSVC未优化的代码有些冗余。 现在我们来看GCC打开优化选项-O3：

Listing 17.13: Optimizing GCC

```
    public f
f       proc near
arg_0 = dword ptr 8
    push ebp
    mov ebp, esp
    mov eax, [ebp+arg_0]
    pop ebp
    or ah, 40h
    and ah, 0FDh
    retn
f       endp
```

代码更短。值得注意的是编译器使用了AH寄存器-EAX寄存器8bit-15bit部分。

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RAX^{x64}							
				EAX			
						AX	
						AH	AL

8086 16位CPU累加器被称为AX，包含两个8位部分-AL（低字节）和AH（高字节）。在80386下所有寄存器被扩展为32位，累加器被命名为EAX，为了保持兼容性，它的老的部分仍可以作为AX/AH/AL寄存器来访问。 因为所有的x86 CPU都兼容于16位CPU，所以老的16位操作码比32位操作码更短。”or ah,40h”指令仅复制3个字节比“or eax,04000h”需要复制5个字节甚至6个字节（如果第一个操作码不是EAX）更合理。

Optimizing GCC and regparm

编译时候开启-O3并且设置regram=3生成的代码会更短。

Listing 19.14: Optimizing GCC

```
        public f
f       proc near
    push ebp
    or ah, 40h
    mov ebp, esp
    and ah, 0FDh
    pop ebp
    retn
f       endp
```

事实上，第一个参数已经被加载到EAX了，所以可以直接使用了。值得注意的是，函数序言（push ebp/mov ebp,esp）和结语（pop ebp）很容易被忽略。GCC并没有优化掉这些代码。更短的代码可以使用内联函数（27）。

19.2.2 ARM + Optimizing Keil + ARM mode

Listing 19.15: Optimizing Keil + ARM mode

```
02 0C C0 E3     BIC     R0, R0, #0x200
01 09 80 E3     ORR     R0, R0, #0x4000
1E FF 2F E1     BX  LR
```

BIC是“逻辑and“类似于x86下的AND。ORR是”逻辑or“类似于x86下的OR。

19.2.3 ARM + Optimizing Keil + thumb mode

Listing 19.16: Optimizing Keil + thumb mode

```
01 21 89 03     MOVS     R1, 0x4000
08 43       ORRS     R0, R1
49 11       ASRS     R1, R1, #5 ; generate 0x200 and place to R1
88 43       BICS     R0, R1
70 47       BX   LR5
```

从0x4000右移生成0x200，采用移位使代码更简洁。

19.2.4 ARM + Optimizing Xcode (LLVM) + ARM mode

Listing 19.17: Optimizing Xcode (LLVM) + ARM mode

```
42 0C C0 E3     BIC     R0, R0, #0x4200
01 09 80 E3     ORR     R0, R0, #0x4000
1E FF 2F E1     BX   LR
```

该代码由LLVM生成，从源码形式上看，看起来更像是：

```
REMOVE_BIT (rt, 0x4200);
SET_BIT (rt, 0x4000);
```

为什么是0x4200?可能是编译器构造的5，可能是编译器编译错误，但生成的代码是可用的。更多编译器异常请参考相关资料（67）。对于thumb模式，优化Xcode(LLVM)生成的代码相似。

19.2.5 ARM: more about the BIC instruction

19.2.6 ARM64: Optimizing GCC (Linaro) 4.9

19.2.7 ARM64: Non-optimizing GCC (Linaro) 4.9

19.2.8 MIPS

19.3 Shifts

C/C++的移位操作通过<<和>>实现。

19.4 设定并请除特定的bit

19.4.1 关于异或的一点

19.4.2 x86

19.4.3 MIPS

19.4.4 ARM

Optimizing Keil 6/2013 (ARM mode)

Optimizing Keil 6/2013 (Thumb mode)

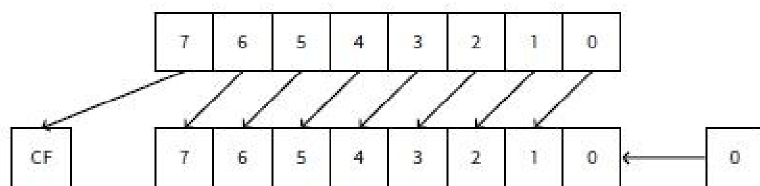
Optimizing GCC 4.6.3 (Raspberry Pi, ARM mode)

19.5 计数bit 来置1

这里有一个例子函数，计算输入变量有多少个位被置为1.

```
#define IS_SET(flag, bit) ((flag) & (bit))
int f(unsigned int a)
{
    int i;
    int rt=0;
    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;
        return rt;
    };
```

在循环中，迭代计数从0到31， `1<<i` 语句将计数从1到0x80000000。1<<i即1左移n位，将包含32位数字所有可能的bit位。每次移位仅有1位被置1，其它位均为0，IS_SET宏将判断a对应的位是否置1。



IS_SET宏就是逻辑与(AND)操作，如果对应的位不为1，则返回0。if条件表达式如果不为0，if()将被触发。

19.5.1 x86

MSVC

MSVC 2010:

Listing 19.18: MSVC 2010

```
_rt$ = -8       ; size = 4
_i$ = -4        ; size = 4
_a$ = 8         ; size = 4
_f PROC
    push ebp
    mov ebp, esp
    sub esp, 8
    mov DWORD PTR _rt$[ebp], 0
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN4@f
$LN3@f:
    mov eax, DWORD PTR _i$[ebp]     ; increment of 1
    add eax, 1
    mov DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp DWORD PTR _i$[ebp], 32      ; 00000020H
    jge SHORT $LN2@f            ; loop finished?
    mov edx, 1
    mov ecx, DWORD PTR _i$[ebp]
    shl edx, cl             ; EDX=EDX<<CL
    and edx, DWORD PTR _a$[ebp]
    je SHORT $LN1@f             ; result of AND instruction was
0?
    ; then skip next instructions
    mov eax, DWORD PTR _rt$[ebp]    ; no, not zero
    add eax, 1 ; increment rt
    mov DWORD PTR _rt$[ebp], eax
$LN1@f:
    jmp SHORT $LN3@f
$LN2@f:
    mov eax, DWORD PTR _rt$[ebp]
    mov esp, ebp
    pop ebp
    ret 0
_f ENDP
```

OllyDbg

GCC

下面是GCC 4.4.1编译的代码： Listing 19.19: GCC 4.4.1

```
        public f
f           proc near
rt      = dword ptr -0Ch
i       = dword ptr -8
arg_0   = dword ptr 8

        push ebp
        mov ebp, esp
        push ebx
        sub esp, 10h
        mov [ebp+rt], 0
        mov [ebp+i], 0
        jmp short loc_80483EF
loc_80483D0:
        mov eax, [ebp+i]
        mov edx, 1
        mov ebx, edx
        mov ecx, eax
        shl ebx, cl
        mov eax, ebx
        and eax, [ebp+arg_0]
        test eax, eax
        jz short loc_80483EB
        add [ebp+rt], 1
loc_80483EB:
        add [ebp+i], 1
loc_80483EF:
        cmp [ebp+i], 1Fh
            jle short loc_80483D0
            mov eax, [ebp+rt]
        add esp, 10h
        pop ebx
        pop ebp
        retn
f           endp
```

19.5.2 x64

Non-optimizing GCC 4.8.2

Optimizing GCC 4.8.2

Optimizing MSVC 2010

Optimizing MSVC 2012

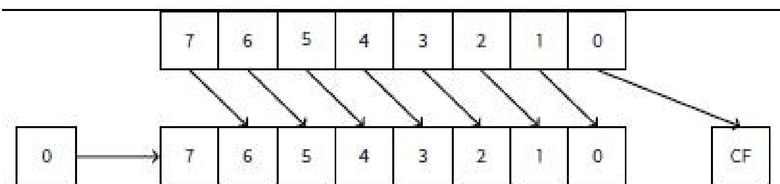
在乘以或者除以2的指数值（1,2,4,8等）时经常使用移位操作。例如：

```
unsigned int f(unsigned int a)
{
    return a/4;
};
```

MSVC 2010: Listing 19.20: MSVC 2010

```
_a$ = 8                     ; size = 4
_f      PROC
        mov eax, DWORD PTR _a$[esp-4]
        shr eax, 2
        ret 0
_f ENDP
```

例子中的SHR（逻辑右移）指令将a值右移2位，最高两位被置0，最低2位被丢弃。实施上丢弃的两位是除法的余数。 SHR作用类似SHL只是移位方向不同。



使用十进制23很好来理解。23除以10，丢弃最后的数字（3是余数），商为2。 与此类似的是乘法。比如乘以4，仅需将数字左移2位，最低两位被置0。就像3乘以100—仅仅在最后补两个0就行了。

19.5.3 ARM + Optimizing Xcode (LLVM) + ARM mode

Listing 19.21: Optimizing Xcode (LLVM) + ARM mode

```
        MOV R1, R0
        MOV R0, #0
        MOV R2, #1
        MOV R3, R0
loc_2E54
        TST R1, R2,LSL R3       ; set flags according to R1 & (R
2<<R3)
        ADD R3, R3, #1      ; R3++
        ADDNE R0, R0, #1        ; if ZF flag is cleared by TST,
R0++
        CMP R3, #32
        BNE loc_2E54
        BX LR
```

TST类似于x86下的TEST指令。 正如我前面提到的(14.2.1)，ARM模式下没有单独的移位指令。对于用作修饰的LSL（逻辑左移）、LSR（逻辑右移）、ASR（算术右移）、ROR（循环右移）和RRX（带扩展的循环右移指令），需要与MOV，TST，CMP，ADD，SUB，RSB结合来使用6。 这些修饰指令被定义，第二个操作数指定移动的位数。 因此“TST R1, R2,LSL R3”指令所做的工作为????1 ∧ (????2 ≪ ????3).

19.5.4 ARM + Optimizing Xcode (LLVM) + thumb-2 mode

几乎一样，只是这里使用LSL.W/TST指令而不是只有TST。因为Thumb模式下TST没有定义修饰符LSL。

```
    MOV      R1, R0
    MOVS     R0, #0
    MOV.W    R9, #1
    MOVS     R3, #0
loc_2F7A
    LSL.W    R2, R9, R3
    TST      R2, R1
    ADD.W    R3, R3, #1
    IT NE
    ADDNE    R0, #1
    CMP      R3, #32
    BNE      loc_2F7A
    BX       LR
```

19.5.5 ARM64 + Optimizing GCC 4.9

19.5.6 ARM64 + Non-optimizing GCC 4.9

19.5.7 MIPS

Non-optimizing GCC

Optimizing GCC

19.6 Conclusion

19.6.1 Check for specific bit (known at compile stage)

19.6.2 Check for specific bit (specified at runtime)

19.6.3 Set specific bit (known at compile stage)

19.6.4 Set specific bit (specified at runtime)

19.6.5 Clear specific bit (known at compile stage)

19.6.6 Clear specific bit (specified at runtime)

19.7 练习

第二十章

用线性同余生成器来产生伪随机数

20.1 x86

20.2 x64

20.3 32-bit ARM

20.4 MIPS

其他线程安全的版本的例子

第二十一章

结构体

C/C++的结构体可以这么定义：它是一组存储在内存中的变量的集合，成员变量类型不要求相同。

21.1 SYSTEMTIME 的例子

让我们看看Win32结构体SYSTEMTIME的定义：

清单21.1: WinBase.h

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

让我们写一个获取当前时间的C程序：

```
#include <windows.h>
#include <stdio.h>
void main()
{
    SYSTEMTIME t;
    GetSystemTime(&t);
    printf ("%04d-%02d-%02d %02d:%02d:%02d",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);
    return;
};
```

反汇编结果如下（MSVC 2010）：

清单21.2: MSVC 2010

```
_t$ = -16 ; size = 16
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 16 ; 00000010H
    lea eax, DWORD PTR _t$[ebp]
    push eax
    call DWORD PTR __imp__GetSystemTime@4
    movzx ecx, WORD PTR _t$[ebp+12] ; wSecond
    push ecx
    movzx edx, WORD PTR _t$[ebp+10] ; wMinute
    push edx
    movzx eax, WORD PTR _t$[ebp+8] ; wHour
    push eax
    movzx ecx, WORD PTR _t$[ebp+6] ; wDay
    push ecx
    movzx edx, WORD PTR _t$[ebp+2] ; wMonth
    push edx
    movzx eax, WORD PTR _t$[ebp] ; wYear
    push eax
    push OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH,
 00H
    call _printf
    add esp, 28 ; 0000001cH
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
```

在本地栈上程序为这个结构体分配了16个字节：这正是sizeof(WORD)*8的大小（因为结构体里有8个WORD）。请注意结构体是由wYear开始的，因此，我们既可以说这是“传给GetSystemTime()函数的，一个指向SYSTEMTIME结构体的指针”，也可以说是它“传递了wYear的指针”。这两种说法是一样的！GetSystemTime()函数会把当前的年份写入指向的WORD指针中，然后把指针向后移动2个字节（译注：WORD大小为2字节），再写入月份，以此类推。

21.1.1 OllyDbg

21.1.2 用结构体代替数组

事实上，结构体的成员其实就是一个个紧贴在一起的变量。我可以用下面的方法来访问SYSTEMTIME结构体，代码如下：

```
#include <windows.h>
#include <stdio.h>
void main()
{
    WORD array[8];
    GetSystemTime (array);
    printf ("%04d-%02d-%02d %02d:%02d:%02d",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /*
 wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /
* wSecond */);
    return;
};
```

编译器会稍稍给出一点警告：

```
systemtime2.c(7) : warning C4133: ’function’ : incompatible type
s - from ’WORD [8]’ to ’LPSYSTEMTIME’
```

不过至少，它会产生如下代码：

清单21.3: MSVC 2010

```
$SG78573 DB '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
_array$ = -16 ; size = 16
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 16 ; 00000010H
    lea eax, DWORD PTR _array$[ebp]
    push eax
    call DWORD PTR __imp__GetSystemTime@4
    movzx ecx, WORD PTR _array$[ebp+12] ; wSecond
    push ecx
    movzx edx, WORD PTR _array$[ebp+10] ; wMinute
    push edx
    movzx eax, WORD PTR _array$[ebp+8] ; wHoure
    push eax
    movzx ecx, WORD PTR _array$[ebp+6] ; wDay
    push ecx
    movzx edx, WORD PTR _array$[ebp+2] ; wMonth
    push edx
    movzx eax, WORD PTR _array$[ebp] ; wYear
    push eax
    push OFFSET $SG78573
    call _printf
    add esp, 28 ; 0000001cH
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
```

当然，它也能一样正常工作！ 一个很有趣的情况是这两次编译结果居然一样，所以光看编译结果，我们还看不出来到底别人用的结构体还是单单用的变量数组。 不过，没几个人会这么无聊的用这种方法写代码，因为这太麻烦了。还有，结构体也有可能会被开发者更改，交换，等等，所以还是用结构体方便。

21.2 让我们用malloc()为结构体分配空间

但是，有时候把结构体放在堆中而不是栈上却更简单：

```
#include <windows.h>
#include <stdio.h>
void main()
{
    SYSTEMTIME *t;
    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));
    GetSystemTime (t);
    printf ("%04d-%02d-%02d %02d:%02d:%02d",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);
    free (t);
    return;
};
```

让我们用优化/Ox编译一下它，看看我们得到什么东西

清单21.4: 优化的MSVC

```
_main PROC
    push esi
    push 16 ; 00000010H
    call _malloc
    add esp, 4
    mov esi, eax
    push esi
    call DWORD PTR __imp__GetSystemTime@4
    movzx eax, WORD PTR [esi+12] ; wSecond
    movzx ecx, WORD PTR [esi+10] ; wMinute
    movzx edx, WORD PTR [esi+8] ; wHour
    push eax
    movzx eax, WORD PTR [esi+6] ; wDay
    push ecx
    movzx ecx, WORD PTR [esi+2] ; wMonth
    push edx
    movzx edx, WORD PTR [esi] ; wYear
    push eax
    push ecx
    push edx
    push OFFSET $SG78833
    call _printf
    push esi
    call _free
    add esp, 32 ; 00000020H
    xor eax, eax
    pop esi
    ret 0
_main ENDP
```

所以，sizeof(SYSTEMTIME) = 16, 这正是malloc所分配的字节数。它返回了刚刚分配的地址空间，这个指针存在EAX寄存器里。然后，这个指针会被移动到ESI结存器中， GetSystemTime()会用它来存储返回值，这也就是为什么这里分配完之后并没有把EAX放到某个地方保存起来，而是直接使用它的原因。

新指令：MOVZX（Move with Zero eXtent， 0扩展移动）。它可以说是和MOVSX基本一样（13.1.1节），但是，它把其他位都设置为0。这是因为printf()需要一个32位的整数，但是我们的结构体里面是WORD，这只有16位厂。这也就是为什么从WORD复制到INT时第16～31位必须清零的原因了。因为，如果不清除的话，剩余位可能有之前操作留下来的干扰数据。

在下面这个例子里面，我可以用WORD数组来重现这个结构：

```
#include <windows.h>
#include <stdio.h>
void main()
{
    WORD *t;
    t=(WORD *)malloc (16);
    GetSystemTime (t);
    printf ("%04d-%02d-%02d %02d:%02d:%02d",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */
);
    free (t);
    return;
};
```

我们得到：

```
$SG78594 DB '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
_main PROC
    push esi
    push 16 ; 00000010H
    call _malloc
    add esp, 4
    mov esi, eax
    push esi
    call DWORD PTR __imp__GetSystemTime@4
    movzx eax, WORD PTR [esi+12]
    movzx ecx, WORD PTR [esi+10]
    movzx edx, WORD PTR [esi+8]
    push eax
    movzx eax, WORD PTR [esi+6]
    push ecx
    movzx ecx, WORD PTR [esi+2]
    push edx
    movzx edx, WORD PTR [esi]
    push eax
    push ecx
    push edx
    push OFFSET $SG78594
    call _printf
    push esi
    call _free
    add esp, 32 ; 00000020H
    xor eax, eax
    pop esi
    ret 0
_main ENDP
```

同样，我们可以看到编译结果和之前一样。个人重申一次，你不应该在写代码的时候用这么晦涩的方法来表达它。

21.3 Unix:结构体tm

21.3.1 linux

在Linux下，我们看看time.h中的tm结构体是什么样子的：

```
#include <stdio.h>
#include <time.h>
void main()
{
    struct tm t;
    time_t unix_time;
    unix_time=time(NULL);
    localtime_r (&unix_time, &t);
    printf ("Year: %d", t.tm_year+1900);
    printf ("Month: %d", t.tm_mon);
    printf ("Day: %d", t.tm_mday);
    printf ("Hour: %d", t.tm_hour);
    printf ("Minutes: %d", t.tm_min);
    printf ("Seconds: %d", t.tm_sec);
};
```

在GCC 4.4.1下编译得到：

清单21.6：GCC 4.4.1

```
main proc near
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    sub esp, 40h
    mov dword ptr [esp], 0 ; first argument for time()
    call time
    mov [esp+3Ch], eax
    lea eax, [esp+3Ch] ; take pointer to what time() returned
    lea edx, [esp+10h] ; at ESP+10h struct tm will begin
    mov [esp+4], edx ; pass pointer to the structure begin
    mov [esp], eax ; pass pointer to result of time()
    call localtime_r
    mov eax, [esp+24h] ; tm_year
    lea edx, [eax+76Ch] ; edx=eax+1900
    mov eax, offset format ; "Year: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    mov edx, [esp+20h] ; tm_mon
    mov eax, offset aMonthD ; "Month: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    mov edx, [esp+1Ch] ; tm_mday
    mov eax, offset aDayD ; "Day: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    mov edx, [esp+18h] ; tm_hour
    mov eax, offset aHourD ; "Hour: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    mov edx, [esp+14h] ; tm_min
    mov eax, offset aMinutesD ; "Minutes: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    mov edx, [esp+10h]
    mov eax, offset aSecondsD ; "Seconds: %d"
    mov [esp+4], edx ; tm_sec
    mov [esp], eax
    call printf
    leave
    retn
main endp
```

可是，IDA并没有为本地栈上变量建立本地变量名。但是因为我们已经学了汇编了，我们也不需要在这么简单的例子里面如此依赖它。

请也注意一下lea edx, [eax+76ch]，这个指令把eax的值加上0x76c，但是并不修改任何标记位。请也参考LEA的相关章节（B.6.2节）

GDB

为了表现出结构体只是一个个的变量连续排列的东西，让我们重新测试一下这个例子，我们看看time.h： 清单18.7 time.h

```
struct tm
{
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
#include <stdio.h>
#include <time.h>
void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wd
ay, tm_yday, tm_isdst;
    time_t unix_time;
    unix_time=time(NULL);
    localtime_r (&unix_time, &tm_sec);
    printf ("Year: %d", tm_year+1900);
    printf ("Month: %d", tm_mon);
    printf ("Day: %d", tm_mday);
    printf ("Hour: %d", tm_hour);
    printf ("Minutes: %d", tm_min);
    printf ("Seconds: %d", tm_sec);
};
```

注：指向tm_sec的指针会传递给localtime_r，或者说第一个“结构体”元素。 编译器会这么警告我们

清单18.8 GCC4.7.3

```
GCC_tm2.c: In function ’main’:
GCC_tm2.c:11:5: warning: passing argument 2 of ’localtime_r’ fro
m incompatible pointer type [
enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected ’struct tm *’ but argu
ment is of type ’int *’
```

但是至少，它会生成这段代码：

清单18.9 GCC 4.7.3

```
main proc near
    var_30 = dword ptr -30h
    var_2C = dword ptr -2Ch
    unix_time = dword ptr -1Ch
    tm_sec = dword ptr -18h
    tm_min = dword ptr -14h
    tm_hour = dword ptr -10h
    tm_mday = dword ptr -0Ch
    tm_mon = dword ptr -8
    tm_year = dword ptr -4
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    sub esp, 30h
    call __main
    mov [esp+30h+var_30], 0 ; arg 0
    mov [esp+30h+unix_time], eax
    lea eax, [esp+30h+tm_sec]
    mov [esp+30h+var_2C], eax
    lea eax, [esp+30h+unix_time]
    mov [esp+30h+var_30], eax
    call localtime_r
    mov eax, [esp+30h+tm_year]
    add eax, 1900
    mov [esp+30h+var_2C], eax
    mov [esp+30h+var_30], offset aYearD ; "Year: %d"
    call printf
    mov eax, [esp+30h+tm_mon]
    mov [esp+30h+var_2C], eax
    mov [esp+30h+var_30], offset aMonthD ; "Month: %d"
    call printf
    mov eax, [esp+30h+tm_mday]
    mov [esp+30h+var_2C], eax
    mov [esp+30h+var_30], offset aDayD ; "Day: %d"
    call printf
    mov eax, [esp+30h+tm_hour]
    mov [esp+30h+var_2C], eax
    mov [esp+30h+var_30], offset aHourD ; "Hour: %d"
    call printf
    mov eax, [esp+30h+tm_min]
    mov [esp+30h+var_2C], eax
    mov [esp+30h+var_30], offset aMinutesD ; "Minutes: %d"
    call printf
    mov eax, [esp+30h+tm_sec]
    mov [esp+30h+var_2C], eax
    mov [esp+30h+var_30], offset aSecondsD ; "Seconds: %d"
    call printf
    leave
    retn
main endp
```

这个代码和我们之前看到的一样，依然无法分辨出源代码是用了结构体还是只是数组而已。

当然这样也是可以运行的，但是实际操作中还是不建议用这种晦涩的方法。因为通常，编译器会在栈上按照声明顺序分配变量空间，但是并不能保证每次都是这样。

还有，其他编译器可能会警告tm_year,tm_mon, tm_mday, tm_hour, tm_min变量而不是tm_sec使用时未初始化。事实上，计算机并不知道调用localtime_r()的时候他们会被自动填充上。

我选择了这个例子来解释是因为他们都是int类型的，而SYSTEMTIME的所有成员是16位的WORD，如果把它们作为本地变量来声明的话，他们会按照32位的边界值来对齐，因此什么都用不了了（因为由于数据对齐，此时GetSystemTime()会把它们错误的填充起来）。请继续读下一节的内容："结构体的成员封装"。

所以，结构体只是把一组变量封装到一个位置上，数据是一个接一个的。我可以说结构体是一个语法糖，因为它只是用来让编译器把一组变量保存在一个地方。但是，我不是编程方面的专家，所以更有可能的是，我可能会误读这个术语。还有，在早期（1972年以前）的时候，C是不支持结构体的。

21.3.2 ARM

ARM+优化Keil+thumb模式

同样的例子： 清单21.10： 优化Keil+thumb模式

```
var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer = -0xC
    PUSH {LR}
    MOVS R0, #0 ; timer
    SUB SP, SP, #0x34
    BL time
    STR R0, [SP,#0x38+timer]
    MOV R1, SP ; tp
    ADD R0, SP, #0x38+timer ; timer
    BL localtime_r
    LDR R1, =0x76C
    LDR R0, [SP,#0x38+var_24]
    ADDS R1, R0, R1
    ADR R0, aYearD ; "Year: %d"
    BL __2printf
    LDR R1, [SP,#0x38+var_28]
    ADR R0, aMonthD ; "Month: %d"
    BL __2printf
    LDR R1, [SP,#0x38+var_2C]
    ADR R0, aDayD ; "Day: %d"
    BL __2printf
    LDR R1, [SP,#0x38+var_30]
    ADR R0, aHourD ; "Hour: %d"
    BL __2printf
    LDR R1, [SP,#0x38+var_34]
    ADR R0, aMinutesD ; "Minutes: %d"
    BL __2printf
    LDR R1, [SP,#0x38+var_38]
    ADR R0, aSecondsD ; "Seconds: %d"
    BL __2printf
    ADD SP, SP, #0x34
    POP {PC}
```

ARM+优化Xcode（LLVM）+thumb-2模式

IDA“碰巧知道”tm结构体（因为IDA“知道”例如localtime_r()这些库函数的参数类型），所以他把这里的结构变量的名字也显示出来了。

```
var_38 = -0x38
var_34 = -0x34
    PUSH {R7,LR}
    MOV R7, SP
    SUB SP, SP, #0x30
    MOVS R0, #0 ; time_t *
    BLX _time
```

```
    ADD R1, SP, #0x38+var_34 ; struct tm *
    STR R0, [SP,#0x38+var_38]
    MOV R0, SP ; time_t *
    BLX _localtime_r
    LDR R1, [SP,#0x38+var_34.tm_year]
    MOV R0, 0xF44 ; "Year: %d"
    ADD R0, PC ; char *
    ADDW R1, R1, #0x76C
    BLX _printf
    LDR R1, [SP,#0x38+var_34.tm_mon]
    MOV R0, 0xF3A ; "Month: %d"
    ADD R0, PC ; char *
    BLX _printf
    LDR R1, [SP,#0x38+var_34.tm_mday]
    MOV R0, 0xF35 ; "Day: %d"
    ADD R0, PC ; char *
    BLX _printf
    LDR R1, [SP,#0x38+var_34.tm_hour]
    MOV R0, 0xF2E ; "Hour: %d"
    ADD R0, PC ; char *
    BLX _printf
    LDR R1, [SP,#0x38+var_34.tm_min]
    MOV R0, 0xF28 ; "Minutes: %d"
    ADD R0, PC ; char *
    BLX _printf
    LDR R1, [SP,#0x38+var_34]
    MOV R0, 0xF25 ; "Seconds: %d"
    ADD R0, PC ; char *
    BLX _printf
    ADD SP, SP, #0x30
    POP {R7,PC}
...
00000000 tm struc ; (sizeof=0x2C, standard type)
00000000 tm_sec DCD ?
00000004 tm_min DCD ?
00000008 tm_hour DCD ?
0000000C tm_mday DCD ?
00000010 tm_mon DCD ?
00000014 tm_year DCD ?
00000018 tm_wday DCD ?
0000001C tm_yday DCD ?
00000020 tm_isdst DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone DCD ? ; offset
0000002C tm ends
```

清单21.11： ARM+优化Xcode（LLVM）+thumb-2模式

21.3.3 MIPS

21.3.4 将结构体作为一组值

21.3.5 讲结构体作为一个32位的数组

21.3.6 讲结构体作为bit的数组

21.4 结构体的成员封装

结构体做的一个重要的事情就是封装了成员，让我们看看简单的例子：

```
#include <stdio.h>
struct s
{
    char a;
    int b;
    char c;
    int d;
};
void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d", s.a, s.b, s.c, s.d);
};
```

如我们所看到的，我们有2个char成员（每个1字节），和两个int类型的数据（每个4字节）。

x86

编译后得到：

```
_s$ = 8 ; size = 16
?f@@YAXUs@@@Z PROC ; f
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _s$[ebp+12]
    push eax
    movsx ecx, BYTE PTR _s$[ebp+8]
    push ecx
    mov edx, DWORD PTR _s$[ebp+4]
    push edx
    movsx eax, BYTE PTR _s$[ebp]
    push eax
    push OFFSET $SG3842
    call _printf
    add esp, 20 ; 00000014H
    pop ebp
    ret 0
?f@@YAXUs@@@Z ENDP ; f
_TEXT ENDS
```

如我们所见，每个成员的地址都按4字节对齐了，这也就是为什么char也会像int一样占用4字节。为什么？因为对齐后对CPU来说更容易读取数据。

但是，这么看明显浪费了一些空间。 让我们能用/Zp1（/Zp[n]代表结构体边界值为n字节）来编译它：

清单18.12: MSVC /Zp1

```
_TEXT SEGMENT
_s$ = 8 ; size = 10
?f@@YAXUs@@@Z PROC ; f
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _s$[ebp+6]
    push eax
    movsx ecx, BYTE PTR _s$[ebp+5]
    push ecx
    mov edx, DWORD PTR _s$[ebp+1]
    push edx
    movsx eax, BYTE PTR _s$[ebp]
    push eax
    push OFFSET $SG3842
    call _printf
    add esp, 20 ; 00000014H
    pop ebp
    ret 0
?f@@YAXUs@@@Z ENDP ; f
```

现在，结构体只用了10字节，而且每个char都占用1字节。我们得到了最小的空间，但是反过来看，CPU却无法用最优化的方式存取这些数据。 可以容易猜到的是，如果这个结构体在很多源代码和对象中被使用的话，他们都需要用同一种方式来编译起来。 除了MSVC /Zp选项，还有一个是#pragma pack编译器选项可以在源码中定义边界值。这个语句在MSVC和GCC中均被支持。 回到SYSTEMTIME结构体中的16位成员，我们的编译器怎么才能把它们按1字节边界来打包？ WinNT.h有这么个代码：

清单18.13:WINNT.H

```
#include "pshpack1.h"
```

和这个：

清单18.14:WINNT.H

```
#include "pshpack4.h" // 4 byte packing is the default
```

文件PshPack1.h看起来像

清单18.15: PSHPACK1.H

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_S
UPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

这就是#pragma pack处理结构体大小的方法。

OllyDbg + fields are packed by default

OllyDbg + fields aligning on 1 byte boundary

21.4.2 ARM

ARM+优化Keil+thumb模式

清单18.16

```
.text:0000003E exit ; CODE XREF: f+16
.text:0000003E 05 B0 ADD SP, SP, #0x14
.text:00000040 00 BD POP {PC}


.text:00000280 f
.text:00000280
.text:00000280 var_18 = -0x18
.text:00000280 a = -0x14
.text:00000280 b = -0x10
.text:00000280 c = -0xC
.text:00000280 d = -8
.text:00000280
.text:00000280 0F B5 PUSH {R0-R3,LR}
.text:00000282 81 B0 SUB SP, SP, #4
.text:00000284 04 98 LDR R0, [SP,#16] ; d
.text:00000286 02 9A LDR R2, [SP,#8] ; b
.text:00000288 00 90 STR R0, [SP]
.text:0000028A 68 46 MOV R0, SP
.text:0000028C 03 7B LDRB R3, [R0,#12] ; c
.text:0000028E 01 79 LDRB R1, [R0,#4] ; a
.text:00000290 59 A0 ADR R0, aADBDCDDD ; "a=%d; b=%d; c=%d; d=%d
"
.text:00000292 05 F0 AD FF BL __2printf
.text:00000296 D2 E6 B exit
```

我们可以回忆到的是，这里它直接用了结构体而不是指向结构体的指针，而且因为ARM里函数的前4个参数是通过寄存器传递的，所以结构体其实是通过R0-R3寄存器传递的。

LDRB指令将内存中的一个字节载入，然后把它扩展到32位，同时也考虑它的符号。这和x86架构的MOVSX（参考13.1.1节）基本一样。这里它被用来传递结构体的a、c两个成员。

还有一个我们可以容易指出来的是，在函数的末尾处，这里它没有使用正常的函数尾该有的指令，而是直接跳转到了另一个函数的末尾！ 的确，这是一个相当不同的函数，而且跟我们的函数没有任何关联。但是，他却有着相同的函数结尾（也许是因为他也有5个本地变量（5 x 4 = 0x14））。而且他就在我们的函数附近（看看地址就知道了）。事实上，函数结尾并不重要，只要函数好好执行就行了嘛。显然，Keil决定要重用另一个函数的一部分，原因就是为了优化代码大小。普通函数结尾需要4字节，而跳转指令只要2个字节。

ARM+优化XCode（LLVM）+thumb-2模式

清单18.17: 优化的Xcode （LLVM）+thumb-2模式

```
var_C = -0xC
    PUSH {R7,LR}
    MOV R7, SP
    SUB SP, SP, #4
    MOV R9, R1 ; b
    MOV R1, R0 ; a
    MOVW R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d
"
    SXTB R1, R1 ; prepare a
    MOVT.W R0, #0
    STR R3, [SP,#0xC+var_C] ; place d to stack for printf()
    ADD R0, PC ; format-string
    SXTB R3, R2 ; prepare c
    MOV R2, R9 ; b
    BLX _printf
    ADD SP, SP, #4
    POP {R7,PC}
```

SXTB（Singned Extend Byte，有符号扩展字节）和x86的MOVSX（见13.1.1节）差不多，但是它不是对内存操作的，而是对一个寄存器操作的，至于剩余的——都一样。

21.4.3 MIPS

21.5 嵌套结构

如果一个结构体里定义了另一个结构体会怎么样？

```
#include <stdio.h>
struct inner_struct
{
    int a;
    int b;
};
struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};
void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d", s.a, s.b,
s.c.a, s.c.b, s.d, s.e);
};
```

在这个例子里，我们把inner_struct放到了outer_struct的abde中间。 让我们在MSVC 2010中编译：

清单18.18： MSVC 2010

```
_s$ = 8 ; size = 24
_f PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _s$[ebp+20] ; e
    push eax
    movsx ecx, BYTE PTR _s$[ebp+16] ; d
    push ecx
    mov edx, DWORD PTR _s$[ebp+12] ; c.b
    push edx
    mov eax, DWORD PTR _s$[ebp+8] ; c.a
    push eax
    mov ecx, DWORD PTR _s$[ebp+4] ; b
    push ecx
    movsx edx, BYTE PTR _s$[ebp] ;a
    push edx
    push OFFSET $SG2466
    call _printf
    add esp, 28 ; 0000001cH
    pop ebp
    ret 0
_f ENDP
```

一个令我们好奇的事情是，看看这个反汇编代码，我们甚至不知道它的体内有另一个结构体！因此，我们可以说，嵌套的结构体，最终都会转化为线性的或者一维的结构。 当然，如果我们把struct inner_struct c;换成struct inner_struct *c（因此这里其实是定义个了一个指针），这个情况下状况则会大为不同。

21.5.1 OllyDbg

21.6 结构体中的位

21.6.1 CPUID 的例子

C/C++中允许给结构体的每一个成员都定义一个准确的位域。如果我们想要节省空间的话，这个对我们来说将是非常有用的。比如，对BOOL来说，1位就足矣了。但是当然，如果我们想要速度的话，必然会浪费点空间。 让我们以CPUID指令为例，这个指令返回当前CPU的信息和特性。 如果EAX在指令执行之前就设置为了1，CPUID将会返回这些内容到EAX中。

3:0	Stepping
7:4	Model
11:8	Family
13:12	Processor Type
19:16	Extended Model
27:20	Extended Family

MSVC 2010有CPUID的宏，但是GCC 4.4.1没有，所以，我们就手动的利用它的内联汇编器为GCC写一个吧。

```
#include <stdio.h>
#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *
d) {
    asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"
(code));
}
#endif
#ifdef _MSC_VER
#include <intrin.h>
#endif
struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};
int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];
    #ifdef _MSC_VER
    __cpuid(b,1);
    #endif
    #ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
    #endif
    tmp=(struct CPUID_1_EAX *)&b[0];
    printf ("stepping=%d", tmp->stepping);
    printf ("model=%d", tmp->model);
    printf ("family_id=%d", tmp->family_id);
    printf ("processor_type=%d", tmp->processor_type);
    printf ("extended_model_id=%d", tmp->extended_model_id);
    printf ("extended_family_id=%d", tmp->extended_family_id);
    return 0;
};
```

之后CPU会填充EAX,EBX,ECX,EDX，这些寄存器的值会通过b[]数组显现出来。接着我们用一个指向CPUID_1_EAX结构体的指针，把它指向b[]数组的EAX值。 换句话说，我们将把32位的INT类型的值当作一个结构体来看。 然后我们就能从结构体中读取数据。

MSVC

让我们在MSVC 2008用/Ox编译一下：

清单18.19: MSVC 2008

```
_b$ = -16 ; size = 16
_main PROC
    sub esp, 16 ; 00000010H
    push ebx
    xor ecx, ecx
    mov eax, 1
    cpuid
    push esi
    lea esi, DWORD PTR _b$[esp+24]
    mov DWORD PTR [esi], eax
    mov DWORD PTR [esi+4], ebx
    mov DWORD PTR [esi+8], ecx
    mov DWORD PTR [esi+12], edx
    mov esi, DWORD PTR _b$[esp+24]
    mov eax, esi
    and eax, 15 ; 0000000fH
    push eax
    push OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call _printf
    mov ecx, esi
    shr ecx, 4
    and ecx, 15 ; 0000000fH
    push ecx
    push OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call _printf
    mov edx, esi
    shr edx, 8
    and edx, 15 ; 0000000fH
    push edx
    push OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
    call _printf
    mov eax, esi
    shr eax, 12 ; 0000000cH
    and eax, 3
    push eax
    push OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
    call _printf
    mov ecx, esi
    shr ecx, 16 ; 00000010H
    and ecx, 15 ; 0000000fH
    push ecx
    push OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
    call _printf
    shr esi, 20 ; 00000014H
    and esi, 255 ; 000000ffH
    push esi
    push OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
    call _printf
    add esp, 48 ; 00000030H
    pop esi
```

```
    xor eax, eax
    pop ebx
    add esp, 16 ; 00000010H
    ret 0
_main ENDP
```

SHR指令将EAX寄存器的值右移位，移出去的值必须被忽略，例如我们会忽略右边的位。AND指令将清除左边不需要的位，换句话说，它处理过后EAX将只留下我们需要的值。

MSVC + OllyDbg

GCC

让我们在GCC4.4.1下用-O3编译。

清单18.20: GCC 4.4.1

```
main proc near ; DATA XREF: _start+17
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    push esi
    mov esi, 1
    push ebx
    mov eax, esi
    sub esp, 18h
    cpuid
    mov esi, eax
    and eax, 0Fh
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aSteppingD ; "stepping=%d"
    mov dword ptr [esp], 1
    call ___printf_chk
    mov eax, esi
    shr eax, 4
    and eax, 0Fh
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aModelD ; "model=%d"
    mov dword ptr [esp], 1
    call ___printf_chk
    mov eax, esi
    shr eax, 8
    and eax, 0Fh
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aFamily_idD ; "family_id=%d"
    mov dword ptr [esp], 1
    call ___printf_chk
    mov eax, esi
```

```
    shr eax, 0Ch
    and eax, 3
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aProcessor_type ; "processor_t
ype=%d"
    mov dword ptr [esp], 1
    call ___printf_chk
    mov eax, esi
    shr eax, 10h
    shr esi, 14h
    and eax, 0Fh
    and esi, 0FFh
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aExtended_model ; "extended_mo
del_id=%d"
    mov dword ptr [esp], 1
    call ___printf_chk
    mov [esp+8], esi
    mov dword ptr [esp+4], offset unk_80486D0
    mov dword ptr [esp], 1
    call ___printf_chk
    add esp, 18h
    xor eax, eax
    pop ebx
    pop esi
    mov esp, ebp
    pop ebp
    retn
main endp
```

几乎一样。只有一个需要注意的地方就是GCC在调用每个printf()之前会把extended_model_id和extended_family_id的计算联合到一块去，而不是把它们分开计算。

21.6.2 将浮点数当作结构体看待

我们已经在FPU（15章）中注意到了float和double两个类型都是有符号的，他们分为符号、有效数字和指数部分。但是我们能直接用上这些位嘛？让我们试一试float。



(S—sign)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>
struct float_as_struct
{
    unsigned int fraction : 23; // fractional part
    unsigned int exponent : 8; // exponent + 0x3FF
    unsigned int sign : 1; // sign bit
};
float f(float _in)
{
    float f=_in;
    struct float_as_struct t;
    assert (sizeof (struct float_as_struct) == sizeof (float));
    memcpy (&t, &f, sizeof (float));
    t.sign=1; // set negative sign
    t.exponent=t.exponent+2; // multiple d by 2^n (n here is 2)
    memcpy (&f, &t, sizeof (float));
    return f;
};
int main()
{
    printf ("%f", f(1.234));
};
```

float_as_struct结构占用了和float一样多的内存空间，也就是4字节，或者说，32位。 现在我们给输入值设置一个负值，然后指数加2，这样我们就能把整个数按照22的值来倍乘，也就是乘以4。 让我们在MSVC2008无优化模式下编译它。

清单18.21: MSVC 2008

```
_t$ = -8 ; size = 4
_f$ = -4 ; size = 4
__in$ = 8 ; size = 4
?f@@YAMM@Z PROC ; f
    push ebp
    mov ebp, esp
    sub esp, 8
    fld DWORD PTR __in$[ebp]
    fstp DWORD PTR _f$[ebp]
    push 4
    lea eax, DWORD PTR _f$[ebp]
    push eax
    lea ecx, DWORD PTR _t$[ebp]
    push ecx
    call _memcpy
    add esp, 12 ; 0000000cH
    mov edx, DWORD PTR _t$[ebp]
    or edx, -2147483648 ; 80000000H - set minus sign
    mov DWORD PTR _t$[ebp], edx
    mov eax, DWORD PTR _t$[ebp]
    shr eax, 23 ; 00000017H - drop significand
    and eax, 255 ; 000000ffH - leave here only exponent
    add eax, 2 ; add 2 to it
    and eax, 255 ; 000000ffH
    shl eax, 23 ; 00000017H - shift result to place of bits 30:2
3
    mov ecx, DWORD PTR _t$[ebp]
    and ecx, -2139095041 ; 807fffffH - drop exponent
    or ecx, eax ; add original value without exponent with new c
alculated exponent
    mov DWORD PTR _t$[ebp], ecx
    push 4
    lea edx, DWORD PTR _t$[ebp]
    push edx
    lea eax, DWORD PTR _f$[ebp]
    push eax
    call _memcpy
    add esp, 12 ; 0000000cH
    fld DWORD PTR _f$[ebp]
    mov esp, ebp
    pop ebp
    ret 0
?f@@YAMM@Z ENDP ; f
```

有点多余。如果用/Ox编译的话，这里就没有memcpy调用了。f变量会被直接使用，但是没有优化的版本看起来会更容易理解一点。 GCC 4.4.1的-O3选项会怎么做？

清单18.22： Gcc 4.4.1

```
; f(float)
public _Z1ff
_Z1ff proc near
var_4 = dword ptr -4
arg_0 = dword ptr 8
    push ebp
    mov ebp, esp
    sub esp, 4
    mov eax, [ebp+arg_0]
    or eax, 80000000h ; set minus sign
    mov edx, eax
    and eax, 807FFFFFh ; leave only significand and exponent in
EAX
    shr edx, 23 ; prepare exponent
    add edx, 2 ; add 2
    movzx edx, dl ; clear all bits except 7:0 in EAX
    shl edx, 23 ; shift new calculated exponent to its place
    or eax, edx ; add new exponent and original value without ex
ponent
    mov [ebp+var_4], eax
    fld [ebp+var_4]
    leave
    retn
    _Z1ff endp
    public main
    main proc near
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    sub esp, 10h
    fld ds:dword_8048614 ; -4.936
    fstp qword ptr [esp+8]
    mov dword ptr [esp+4], offset asc_8048610 ; "%f
"
    mov dword ptr [esp], 1
    call ___printf_chk
    xor eax, eax
    leave
    retn
main endp
```

f()函数基本可以理解，但是有趣的是，GCC可以在编译阶段就通过我们这堆大杂烩一样的代码计算出f(1.234)的值，从而会把他当作参数直接给printf()。

21.7 练习

第二十二章

联合体

22.1 伪随机数生成器的例子

如果我们需要0～1的随机浮点数，最简单的方法就是用PRNG（伪随机数发生器），比如马特赛特旋转演算法可以生成一个随机的32位的DWORD。然后我们可以把这个值转为FLOAT类型，然后除以RAND_MAX（我们的例子是0xFFFFFFFF），这样，我们得到的将是0..1区间的数。但是如我们所知道的是，除法很慢。我们是否能摆脱它呢？就像我们用乘法做除法一样（14章）。让我们想想浮点数由什么构成：符号位、有效数字位、指数位。我们只需要在这里面存储一些随机的位就好了。指数不能变成0（在本例里面数字会不正常），所以我们存储0111111到指数里面，这意味着指数位将是1。然后，我们用随机位填充有效数字位，然后把符号位设置为0（正数）。生成的数字将在1-2的间隔中生成，所以我们必须从里面再减去1。我例子里面是最简单的线性同余随机数生成器，生成32位（译注：32-bit比特位，非数字位）的数字。PRNG将会用UNIX时间戳来初始化。然后，我们会把float类型当作联合体（union）来处理，这是一个C/C++的结构。它允许我们把一片内存里面各种不同类型的数据联合覆盖到一起用。在我们的例子里，我们可以创建一个union，然后通过float或者uint32_t来访问它。因此，这只是一个小技巧，而且是很脏的技巧。

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>
union uint32_t_float
{
    uint32_t i;
    float f;
};
// from the Numerical Recipes book
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;
int main()
{
    uint32_t_float tmp;
    uint32_t RNG_state=time(NULL); // initial seed
    for (int i=0; i<100; i++)
    {
        RNG_state=RNG_state*RNG_a+RNG_c;
        tmp.i=RNG_state & 0x007fffff | 0x3F800000;
        float x=tmp.f-1;
        printf ("%f", x);
    };
    return 0;
};
```

22.1.1 x86

清单19.1: MSVC 2010 （/Ox）

```
$SG4232 DB '%f', 0aH, 00H
__real@3ff0000000000000 DQ 03ff0000000000000r ; 1
tv140= -4 ; size = 4
_tmp$= -4 ; size = 4
_main PROC
    push ebp
    mov ebp, esp
    and esp, -64 ; ffffffc0H
    sub esp, 56 ; 00000038H
    push esi
    push edi
    push 0
    call __time64
    add esp, 4
    mov esi, eax
    mov edi, 100 ; 00000064H
$LN3@main:
    ; let's generate random 32-bit number
    imul esi, 1664525 ; 0019660dH
    add esi, 1013904223 ; 3c6ef35fH
    mov eax, esi
    ; leave bits for significand only
    and eax, 8388607 ; 007fffffH
    ; set exponent to 1
    or eax, 1065353216 ; 3f800000H
    ; store this value as int
    mov DWORD PTR _tmp$[esp+64], eax
    sub esp, 8
    ; load this value as float
    fld DWORD PTR _tmp$[esp+72]
    ; subtract one from it
    fsub QWORD PTR __real@3ff0000000000000
    fstp DWORD PTR tv140[esp+72]
    fld DWORD PTR tv140[esp+72]
    fstp QWORD PTR [esp]
    push OFFSET $SG4232
    call _printf
    add esp, 12 ; 0000000cH
    dec edi
    jne SHORT $LN3@main
    pop edi
    xor eax, eax
    pop esi
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
_TEXT ENDS
END
```

GCC也生成了非常相似的代码。

22.1.2 MIPS

22.1.3 ARM (ARM mode)

22.2计算器的精度

22.2.1 x86

22.2.2 ARM64

22.2.3 MIPS

22.2.4 Conclusion

22.3 快速开方计算

第二十三章

指向函数的指针

函数指针是指向函数的指针，和其他指针一样，只是该指针指向函数代码段的开始地址。函数指针经常用作回调1。

典型的例子如下：

```
C标准库的 qsort()2, aexit()3；
*NIX OS的信号机制；
线程启动：CreateThread()(Win32)，pthread_create()(POSIX)；
其他更多的Win32函数，比如EnumChildWindows()5。
qsort()函数是C/C++标准库快速排序函数。该函数能够排序任意类型的数据。qsort
()调用比较函数。
```

比较函数被定义为如下形式：

```
int (*compare)(const void *, const void *)
```

我们稍作修改：

```
/* ex3 Sorting ints with qsort */
#include <stdio.h>
#include <stdlib.h>

int comp(const void * _a, const void * _b)
{
    const int *a=(const int *)_a;
    const int *b=(const int *)_b;

    if (*a==*b)
        return 0;
    else
        if (*a < *b)
            return -1;
        else
            return 1;
}

int main(int argc, char* argv[])
{
    int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-1000
00};
    int i;
    /* Sort the array */
    qsort(numbers,10,sizeof(int),comp) ;
    for (i=0;i<9;i++)
        printf("Number = %d",numbers[ i ]) ;
    return 0;
}
```

23.1 MSVC

MSVC2010 /Ox选项编译：

Listing 20.1: Optimizing MSVC 2010: /Ox /GS- /MD

```
__a$ = 8                                                ; size = 4
__b$ = 12                                               ; size = 4
_comp   PROC
        mov     eax, DWORD PTR __a$[esp-4]
        mov     ecx, DWORD PTR __b$[esp-4]
        mov     eax, DWORD PTR [eax]
        mov     ecx, DWORD PTR [ecx]
        cmp     eax, ecx
        jne     SHORT $LN4@comp
        xor     eax, eax
        ret     0
$LN4@comp:
        xor     edx, edx
        cmp     eax, ecx
        setge   dl
        lea     eax, DWORD PTR [edx+edx-1]
        ret     0
_comp ENDP

_numbers$ = -40                                         ; size = 40
_argc$ = 8                                              ; size = 4
_argv$ = 12                                             ; size = 4
_main   PROC
        sub     esp, 40                                 ; 00000028H
        push    esi
        push    OFFSET _comp
        push    4
        lea     eax, DWORD PTR _numbers$[esp+52]
        push    10 ; 0000000aH
        push    eax
        mov     DWORD PTR _numbers$[esp+60], 1892   ; 00000764H
        mov     DWORD PTR _numbers$[esp+64], 45     ; 0000002dH
        mov     DWORD PTR _numbers$[esp+68], 200    ; 000000c8H
        mov     DWORD PTR _numbers$[esp+72], -98    ; ffffff9eH
        mov     DWORD PTR _numbers$[esp+76], 4087   ; 00000ff7H
        mov     DWORD PTR _numbers$[esp+80], 5
        mov     DWORD PTR _numbers$[esp+84], -12345 ; ffffcfc7H
        mov     DWORD PTR _numbers$[esp+88], 1087   ; 0000043fH
        mov     DWORD PTR _numbers$[esp+92], 88     ; 00000058H
        mov     DWORD PTR _numbers$[esp+96], -100000 ; fffe7960H
        call    _qsort
        add     esp, 16 ; 00000010H
...
```

第四个参数传递了一个地址标签_comp，指向了comp()函数。

我们来看MSVCR80.DLL（包含C标准库函数的MSVC DLL模块）里该函数的内部调用：

Listing 20.2: MSVCR80.DLL

```
.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsign
ed int, int (__cdecl *)(const void *, const void *))
.text:7816CBF0          public _qsort
.text:7816CBF0 _qsort   proc near
.text:7816CBF0
.text:7816CBF0 lo       = dword ptr -104h
.text:7816CBF0 hi       = dword ptr -100h
.text:7816CBF0 var_FC   = dword ptr -0FCh
.text:7816CBF0 stkptr   = dword ptr -0F8h
.text:7816CBF0 lostk    = dword ptr -0F4h
.text:7816CBF0 histk    = dword ptr -7Ch
.text:7816CBF0 base     = dword ptr 4
.text:7816CBF0 num      = dword ptr 8
.text:7816CBF0 width    = dword ptr 0Ch
.text:7816CBF0 comp     = dword ptr 10h
.text:7816CBF0
.text:7816CBF0 sub      esp, 100h
....
.text:7816CCE0 loc_7816CCE0:                ; CODE XREF: _qsort+
B1
.text:7816CCE0          shr     eax, 1
.text:7816CCE2          imul    eax, ebp
.text:7816CCE5          add     eax, ebx
.text:7816CCE7          mov     edi, eax
.text:7816CCE9          push    edi
.text:7816CCEA          push    ebx
.text:7816CCEB          call    [esp+118h+comp]
.text:7816CCF2          add     esp, 8
.text:7816CCF5          test    eax, eax
.text:7816CCF7          jle     short loc_7816CD04
```

第四个参数comp传递函数指针，comp()有两个参数，参数被检测后才执行。

这种使用函数指针的方式有一定的风险。第一种原因是如果你用qsort()调用了错误的函数指针，可能造成程序崩溃，并且这个错误很难被发现。

第二个原因是即使回调函数类型完全正确，使用错误的参数调用函数可能会导致更严重的问题。进程崩溃不是最大的问题，最大的问题是崩溃的原因—编译器很难发现这种潜在的问题。

23.1.1 MSVC + OllyDbg

我们在OD中加载我们的例子，并在comp()函数下断点。

我们可以看到第一次comp()调用时是如何比较的：fig.20.1.OD代码窗口显示了比较的值。我们还可以看到SP指向的RA地址在qsort()函数空间里（实际上位于MSVCR100.DLL）。

按F8直到函数返回到qsort()函数：fig20.2.这里比较函数被调用。

第二次调用comp()—当前比较的值不相同：fig203。

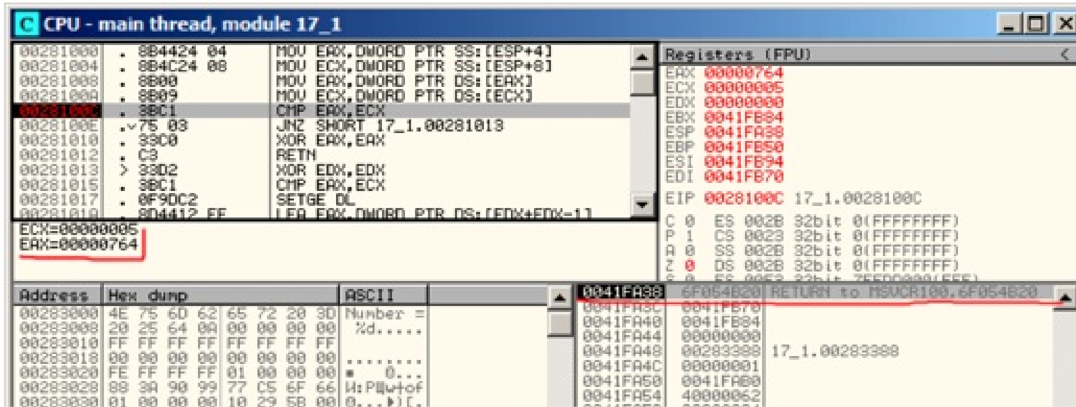


Figure 20.1: OllyDbg: first call of comp()

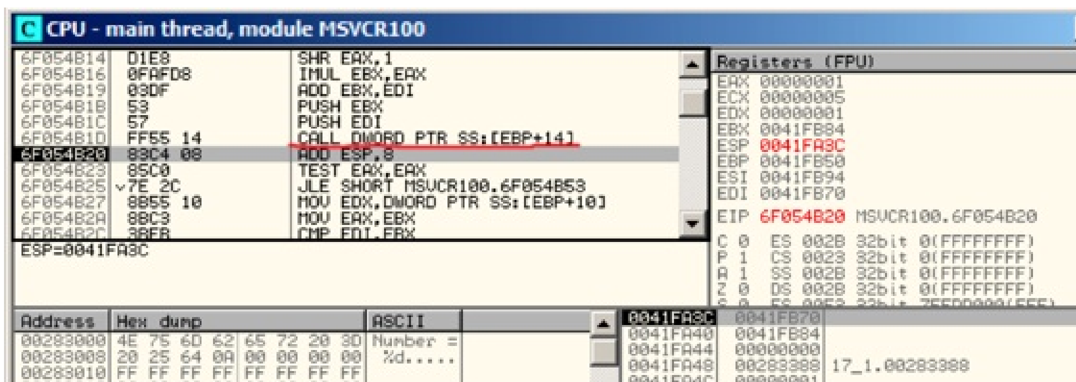


Figure 20.2: OllyDbg: the code in qsort() right a_er comp() call

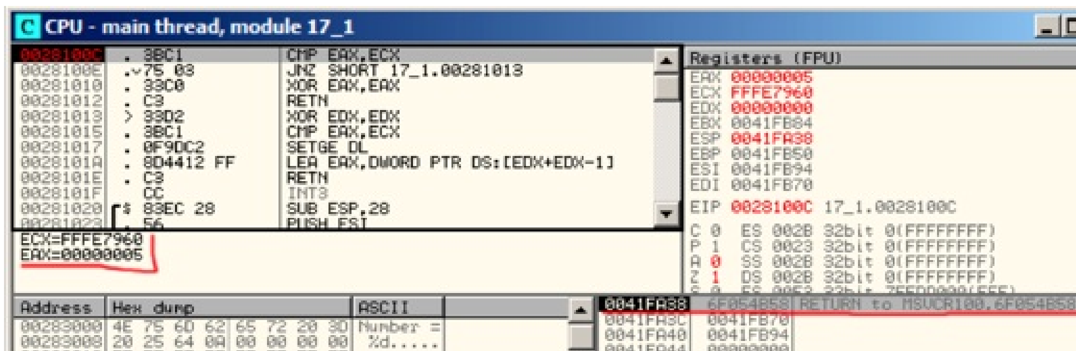


Figure 20.3: OllyDbg: second call of comp()

23.1.2 MSVC + tracer

我们来看成对比较，来对10个数字进行排序：1892, 45, 200, -98, 4087, 5, -12345, 1087, 88,-100000.

我们找到comp()函数中的CMP指令地址，并在其地址0x0040100C上设置断点。

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

断点中断是的寄存器地址：

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

过滤EAX和ECX得到：

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xfffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0xffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x000000c8 ECX=0x00000ff7
EAX=0x0000002d ECX=0x00000ff7
EAX=0x0000043f ECX=0x00000ff7
EAX=0x00000058 ECX=0x00000ff7
EAX=0x00000764 ECX=0x00000ff7
EAX=0x000000c8 ECX=0x00000764
EAX=0x0000002d ECX=0x00000764
EAX=0x0000043f ECX=0x00000764
EAX=0x00000058 ECX=0x00000764
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000043f ECX=0x000000c8
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000002d ECX=0x00000058
```

有34对。因此快速排序算法对10个数字排序需要34此对比操作。

20.1.3 MSVC + tracer (code coverage)

我们使用跟踪特性收集寄存器的值并在IDA中查看。

跟踪comp()函数所有指令：

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

IDA加载.idc脚本：fig20.4。

IDA给出了函数名字(PtFuncCompare)—IDA认为该函数指针被传递给qsort()。

可以看到a和b指向数组不同的位置，并且相差4-32bit的字节数。

0x401010 和 0x401012之间的指令从没有被执行：事实上comp()从来不返回0，因为没有相等的元素。



Figure 20.4: tracer and IDA. N.B.: some values are cutted at right

23.2 GCC

没有太大的不同：

Listing 20.3: GCC

```
lea eax, [esp+40h+var_28]
mov [esp+40h+var_40], eax
mov [esp+40h+var_28], 764h
mov [esp+40h+var_24], 2Dh
mov [esp+40h+var_20], 0C8h
mov [esp+40h+var_1C], 0FFFFFF9Eh
mov [esp+40h+var_18], 0FF7h
mov [esp+40h+var_14], 5
mov [esp+40h+var_10], 0FFFFCFC7h
mov [esp+40h+var_C], 43Fh
mov [esp+40h+var_8], 58h
mov [esp+40h+var_4], 0FFFE7960h
mov [esp+40h+var_34], offset comp
mov [esp+40h+var_38], 4
mov [esp+40h+var_3C], 0Ah
call _qsort
```

comp() 函数：

```
        public comp
comp    proc near
arg_0   = dword ptr 8
arg_4   = dword ptr 0Ch
        push ebp
        mov ebp, esp
        mov eax, [ebp+arg_4]
        mov ecx, [ebp+arg_0]
        mov edx, [eax]
        xor eax, eax
        cmp [ecx], edx
        jnz short loc_8048458
        pop ebp
        retn
loc_8048458:
        setnl al
        movzx eax, al
        lea eax, [eax+eax-1]
        pop ebp
        retn
comp    endp
```

qsort()的实现在libc.so里，它实际上是qsort_r()的封装。

我们通过传递函数指针然后调用快速排序：

Listing 20.4: (file libc.so.6, glibc version—2.10.1)

```
.text:0002DDF6 mov edx, [ebp+arg_10]
.text:0002DDF9 mov [esp+4], esi
.text:0002DDFD mov [esp], edi
.text:0002DE00 mov [esp+8], edx
.text:0002DE04 call [ebp+arg_C]
...
```

23.2.1 GCC + GDB (with source code)

因为我们有例子的C源代码，我们能在行数(11—第一次比较的地方)设置断点(b)。编译例子的时候使用了带有调试信息的选项(-g)，当前可以查看地址及行号，也可以打印变量(p):调试信息包含寄存器和变量值信息。

我们查看堆栈(bt)，看到glibc使用的中间函数msort_with_tmp()。

Listing 20.5: GDB session

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licen
ses/gpl.html>
This is free software: you are free to change and redistribute i
t.
There is NO WARRANTY, to the extent permitted by law. Type "show
 copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out
Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17
_1.c:11
11 if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.
Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17
_1.c:11
11 if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0 comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1 0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry
=0xbffff0f8, n=n@entry=2)
at msort.c:65
#2 0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c
) at msort.c:45
#3 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n
=n@entry=5) at msort.c:53
#4 0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c
) at msort.c:45
#5 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n
=n@entry=10) at msort.c:53
#6 0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07
c) at msort.c:45
#7 __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4
, cmp=cmp@entry=0x804844d <comp
>,
arg=arg@entry=0x0) at msort.c:297
#8 0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x8048
44d <comp>) at msort.c:307
#9 0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
```

```
(gdb)
```

23.2.2 GCC + GDB (no source code)

更多时候我们没有源码，我们可以反汇编comp()函数（disas），找到CMP指令地址并设置断点(b)。每次中断后，dump所有寄存器的值(info registers)，堆栈信息(bt)，但是没有comp()函数对应的行号信息。

Listing 20.6: GDB session

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licen
ses/gpl.html>
This is free software: you are free to change and redistribute i
t.
There is NO WARRANTY, to the extent permitted by law. Type "show
 copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...(no debugging
symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
0x0804844d <+0>: push ebp
0x0804844e <+1>: mov ebp,esp
0x08048450 <+3>: sub esp,0x10
0x08048453 <+6>: mov eax,DWORD PTR [ebp+0x8]
0x08048456 <+9>: mov DWORD PTR [ebp-0x8],eax
0x08048459 <+12>: mov eax,DWORD PTR [ebp+0xc]
0x0804845c <+15>: mov DWORD PTR [ebp-0x4],eax
0x0804845f <+18>: mov eax,DWORD PTR [ebp-0x8]
0x08048462 <+21>: mov edx,DWORD PTR [eax]
0x08048464 <+23>: mov eax,DWORD PTR [ebp-0x4]
0x08048467 <+26>: mov eax,DWORD PTR [eax]
0x08048469 <+28>: cmp edx,eax
0x0804846b <+30>: jne 0x8048474 <comp+39>
0x0804846d <+32>: mov eax,0x0
0x08048472 <+37>: jmp 0x804848e <comp+65>
0x08048474 <+39>: mov eax,DWORD PTR [ebp-0x8]
0x08048477 <+42>: mov edx,DWORD PTR [eax]
0x08048479 <+44>: mov eax,DWORD PTR [ebp-0x4]
0x0804847c <+47>: mov eax,DWORD PTR [eax]
0x0804847e <+49>: cmp edx,eax
0x08048480 <+51>: jge 0x8048489 <comp+60>
0x08048482 <+53>: mov eax,0xffffffff
```

```
0x08048487 <+58>: jmp 0x804848e <comp+65>
0x08048489 <+60>: mov eax,0x1
0x0804848e <+65>: leave
0x0804848f <+66>: ret
End of assembler dump.
(gdb) b *0x08048469
Breakpoint 1 at 0x8048469
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax 0x2d 45
ecx 0xbffff0f8 -1073745672
edx 0x764 1892
ebx 0xb7fc0000 -1208221696
esp 0xbfffeeb8 0xbfffeeb8
ebp 0xbfffeec8 0xbfffeec8
esi 0xbffff0fc -1073745668
edi 0xbffff010 -1073745904
eip 0x8048469 0x8048469 <comp+28>
eflags 0x286 [ PF SF IF ]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax 0xff7 4087
ecx 0xbffff104 -1073745660
edx 0xffffff9e -98
ebx 0xb7fc0000 -1208221696
esp 0xbfffee58 0xbfffee58
ebp 0xbfffee68 0xbfffee68
esi 0xbffff108 -1073745656
edi 0xbffff010 -1073745904
eip 0x8048469 0x8048469 <comp+28>
eflags 0x282 [ SF IF ]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51
(gdb) c
Continuing.
Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
```

```
eax 0xffffff9e -98
ecx 0xbffff100 -1073745664
edx 0xc8 200
ebx 0xb7fc0000 -1208221696
esp 0xbfffeeb8 0xbfffeeb8
ebp 0xbfffeec8 0xbfffeec8
esi 0xbffff104 -1073745660
edi 0xbffff010 -1073745904
eip 0x8048469 0x8048469 <comp+28>
eflags 0x286 [ PF SF IF ]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51
(gdb) bt
#0 0x08048469 in comp ()
#1 0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry
=0xbffff0f8, n=n@entry=2)
at msort.c:65
#2 0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c
) at msort.c:45
#3 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n
=n@entry=5) at msort.c:53
#4 0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c
) at msort.c:45
#5 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n
=n@entry=10) at msort.c:53
#6 0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07
c) at msort.c:45
#7 __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4
, cmp=cmp@entry=0x804844d <comp
>,
arg=arg@entry=0x0) at msort.c:297
#8 0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x8048
44d <comp>) at msort.c:307
#9 0x0804850d in main ()
```

第二十四章

在32位环境中的64位值

在32位环境中的通用寄存器是32位的，所以64位值转化为一对32位值。

24.1 返回64位的值

24.1.1 x86

24.1.2 ARM

24.1.3 MIPS

24.2参数的传递，加法，减法

```
#include <stdint.h>
uint64_t f1 (uint64_t a, uint64_t b)
{
        return a+b;
};
void f1_test ()
{
#ifdef __GNUC__
        printf ("%lld", f1(12345678901234, 23456789012345));
#else
        printf ("%I64d", f1(12345678901234, 23456789012345));
#endif
};
uint64_t f2 (uint64_t a, uint64_t b)
{
        return a-b;
};
```

24.2.1 x86

代码 21.1: MSVC 2012 /Ox /Ob1

```
_a$ = 8                                                  ; size = 8
_b$ = 16                                                 ; size = 8
_f1     PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        adc     edx, DWORD PTR _b$[esp]
        ret     0
_f1     ENDP

_f1_test    PROC
        push        5461                                 ; 00001555H
        push        1972608889                           ; 75939f79H
        push        2874                                 ; 00000b3aH
        push        1942892530                           ; 73ce2ff2H
        call        _f1
        push        edx
        push        eax
        push        OFFSET $SG1436 ; '%I64d', 0aH, 00H
        call        _printf
        add         esp, 28                              ; 0000001cH
        ret     0
_f1_test    ENDP
_f2     PROC
        mov     eax, DWORD PTR _a$[esp-4]
        sub     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        sbb     edx, DWORD PTR _b$[esp]
        ret     0
_f2     ENDP
```

我们可以看到在函数f1_test()中每个64位值转化为2个32位值，高位先转，然后是低位。加法和减法也是如此。

当进行加法操作时，低32位部分先做加法。如果相加过程中产生进位，则设置CF标志。下一步通过ADC指令加上高位部分，如果CF置1了就增加1。

减法操作也是如此。第一个SUB操作也会导致CF标志的改变，并在随后的SBB操作中检查：如果CF置1了，那么最终结果也会减去1。

在32位环境中，64位的值是从EDX:EAX这一对寄存器的函数中返回的。可以很容易看出f1()函数是如何转化为printf()函数的。

代码 21.2: GCC 4.8.1 -O1 -fno-inline

```
_f1:
        mov     eax, DWORD PTR [esp+12]
        mov     edx, DWORD PTR [esp+16]
        add     eax, DWORD PTR [esp+4]
        adc     edx, DWORD PTR [esp+8]
        ret

_f1_test:
        sub     esp, 28
        mov     DWORD PTR [esp+8], 1972608889          ; 75939f
79H
        mov     DWORD PTR [esp+12], 5461               ; 000015
55H
        mov     DWORD PTR [esp], 1942892530            ; 73ce2f
f2H
        mov     DWORD PTR [esp+4], 2874                ; 00000b
3aH
        call    _f1
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp+8], edx
        mov     DWORD PTR [esp], OFFSET FLAT:LC0       ; "%lld1
2"
        call    _printf
        add     esp, 28
        ret

_f2:
        mov     eax, DWORD PTR [esp+4]
        mov     edx, DWORD PTR [esp+8]
        sub     eax, DWORD PTR [esp+12]
        sbb     edx, DWORD PTR [esp+16]
        ret
```

GCC代码也是如此。

24.2.2 ARM

24.2.3 MIPS

21.2 乘法，除法

```
#include <stdint.h>
uint64_t f3 (uint64_t a, uint64_t b)
{
        return a*b;
};
uint64_t f4 (uint64_t a, uint64_t b)
{
        return a/b;
};
uint64_t f5 (uint64_t a, uint64_t b)
{
        return a % b;
};
```

24.3.1 x86

代码 21.3: MSVC 2012 /Ox /Ob1

```
_a$ = 8                                      ; size = 8
_b$ = 16                                     ; size = 8
_f3     PROC
        push        DWORD PTR _b$[esp]
        push        DWORD PTR _b$[esp]
        push        DWORD PTR _a$[esp+8]
        push        DWORD PTR _a$[esp+8]
        call        __allmul ; long long multiplication
        ret         0
_f3     ENDP
_a$ = 8                                      ; size = 8
_b$ = 16                                     ; size = 8
_f4     PROC
        push        DWORD PTR _b$[esp]
        push        DWORD PTR _b$[esp]
        push        DWORD PTR _a$[esp+8]
        push        DWORD PTR _a$[esp+8]
        call        __aulldiv ; unsigned long long division
        ret         0
_f4     ENDP
_a$ = 8                                      ; size = 8
_b$ = 16                                     ; size = 8
_f5     PROC
        push        DWORD PTR _b$[esp]
        push        DWORD PTR _b$[esp]
        push        DWORD PTR _a$[esp+8]
        push        DWORD PTR _a$[esp+8]
        call        __aullrem ; unsigned long long remainder
        ret         0
_f5     ENDP
```

乘法和除法是更为复杂的操作，一般来说，编译器会嵌入库函数的calls来使用。

部分函数的意义：可参见附录E。

Listing 21.4: GCC 4.8.1 -O3 -fno-inline

```
_f3:
        push        ebx
        mov         edx, DWORD PTR [esp+8]
        mov         eax, DWORD PTR [esp+16]
        mov         ebx, DWORD PTR [esp+12]
        mov         ecx, DWORD PTR [esp+20]
        imul        ebx, eax
        imul        ecx, edx
        mul         edx
        add         ecx, ebx
        add         edx, ecx
        pop         ebx
        ret
_f4:
        sub         esp, 28
        mov         eax, DWORD PTR [esp+40]
        mov         edx, DWORD PTR [esp+44]
        mov         DWORD PTR [esp+8], eax
        mov         eax, DWORD PTR [esp+32]
        mov         DWORD PTR [esp+12], edx
        mov         edx, DWORD PTR [esp+36]
        mov         DWORD PTR [esp], eax
        mov         DWORD PTR [esp+4], edx
        call        ___udivdi3 ; unsigned division
        add         esp, 28
        ret
_f5:
        sub         esp, 28
        mov         eax, DWORD PTR [esp+40]
        mov         edx, DWORD PTR [esp+44]
        mov         DWORD PTR [esp+8], eax
        mov         eax, DWORD PTR [esp+32]
        mov         DWORD PTR [esp+12], edx
        mov         edx, DWORD PTR [esp+36]
        mov         DWORD PTR [esp], eax
        mov         DWORD PTR [esp+4], edx
        call        ___umoddi3 ; unsigned modulo
        add         esp, 28
        ret
```

GCC的做法几乎一样，但是乘法代码内联在函数中，可认为这样更有效。

GCC有一些不同的库函数：参见附录D

24.3.2 ARM

24.3.3 MIPS

21.3 右位移

```
#include <stdint.h>
uint64_t f6 (uint64_t a)
{
        return a>>7;
};
```

24.4.1 x86

代码 21.5: MSVC 2012 /Ox /Ob1

```
_a$ = 8                                     ; size = 8
_f6     PROC
        mov     eax, DWORD PTR _a$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        shrd    eax, edx, 7
        shr     edx, 7
        ret     0
_f6     ENDP
```

代码 21.6: GCC 4.8.1 -O3 -fno-inline

```
_f6:
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+4]
        shrd        eax, edx, 7
        shr     edx, 7
        ret
```

右移也是分成两步完成：先移低位，然后移高位。但是低位部分通过指令SHRD移动，它将EDX的值移动7位，并从EAX借来1位，也就是从高位部分。而高位部分通过更受欢迎的指令SHR移动：的确，高位释放出来的位置用0填充。

24.4.2 ARM

24.4.3 MIPS

24.5从32位值转化为64位值

24.5.1 x86

24.5.2 ARM

24.5.3 MIPS

```
#include <stdint.h>
int64_t f7 (int64_t a, int64_t b, int32_t c)
{
        return a*b+c;
};

int64_t f7_main ()
{
        return f7(12345678901234, 23456789012345, 12345);
};
```

代码 21.7: MSVC 2012 /Ox /Ob1

```
_a$ = 8                                  ; size = 8
_b$ = 16                                 ; size = 8
_c$ = 24                                 ; size = 4
_f7     PROC
        push        esi
        push        DWORD PTR _b$[esp+4]
        push        DWORD PTR _b$[esp+4]
        push        DWORD PTR _a$[esp+12]
        push        DWORD PTR _a$[esp+12]
        call        __allmul ; long long multiplication
        mov         ecx, eax
        mov         eax, DWORD PTR _c$[esp]
        mov         esi, edx
        cdq                 ; input: 32-bit value in EAX; output
: 64-bit value in EDX:EAX
        add         eax, ecx
        adc         edx, esi
        pop         esi
        ret         0
_f7     ENDP

_f7_main PROC
        push        12345                ; 00003039H
        push        5461                 ; 00001555H
        push        1972608889           ; 75939f79H
        push        2874                 ; 00000b3aH
        push        1942892530           ; 73ce2ff2H
        call        _f7
        add     esp, 20                  ; 00000014H
        ret     0
_f7_main ENDP
```

这里我们有必要将有符号的32位值从c转化为有符号的64位值。无符号值的转化简单了当：所有的高位部分全部置0。但是这样不适合有符号的数据类型：符号标志应复制到结果中的高位部分。这里用到的指令是CDQ，它从EAX中取出数值，将其变为64位并存放到EDX:EAX这一对寄存器中。换句话说，指令CDQ从EAX中获取符号（通过EAX中最重要的位），并根据它来设置EDX中所有位为0还是为1。它的操作类似于指令MOVSX（13.1.1）。

代码 21.8: GCC 4.8.1 -O3 -fno-inline

```
_f7:
        push        edi
        push        esi
        push        ebx
        mov         esi, DWORD PTR [esp+16]
        mov         edi, DWORD PTR [esp+24]
        mov         ebx, DWORD PTR [esp+20]
        mov         ecx, DWORD PTR [esp+28]
        mov         eax, esi
        mul         edi
        imul        ebx, edi
        imul        ecx, esi
        mov         esi, edx
        add         ecx, ebx
        mov         ebx, eax
        mov         eax, DWORD PTR [esp+32]
        add         esi, ecx
        cdq             ; input: 32-bit value in EAX; output: 64
-bit value in EDX:EAX
        add         eax, ebx
        adc         edx, esi
        pop         ebx
        pop         esi
        pop         edi
        ret
_f7_main:
        sub         esp, 28
        mov         DWORD PTR [esp+16], 12345                 ; 00
003039H
        mov         DWORD PTR [esp+8], 1972608889             ; 75
939f79H
        mov         DWORD PTR [esp+12], 5461                  ; 00
001555H
        mov         DWORD PTR [esp], 1942892530               ; 73
ce2ff2H
        mov         DWORD PTR [esp+4], 2874                   ; 00
000b3aH
        call
_f7
        add         esp, 28
        ret
```

GCC生成的汇编代码跟MSVC一样，但是在函数中内联乘法代码。 更多：32位值在16位环境中（30.4）

第二十五章

SIMD

SIMD是Single Instruction, Multiple Data的首字母。简单说就是单指令多数据流。

就像FPU，FPU看起来更像独立于x86处理器。

SIMD开始于MMX x86。8个新的64位寄存器MM0-MM7被添加。

每个MMX寄存器包含2个32-bit值/4个16-bit值/8字节。比如可以通过一次添加两个值到MMX寄存器来添加8个8-bit（字节）。

一个简单的例子就是图形编辑器，将图像表示为一个二维数组，当用户改变图像的亮度，编辑器必须添加每个像素的差值。为了简单起见，将每个像素定义为一个8位字节，就可以同时改变8个像素的亮度。

当使用MMX的时候，这些寄存器实际上位于FPU寄存器。所以可以同时使用FPU和MMX寄存器。有人可能会认为，intel基于晶体管保存，事实上，这种共生关系的原因是：老的操作系统不知道额外的CPU寄存器，上下文切换是不会保存这些寄存器，可以节省FPU寄存器。这样激活MMX的CPU+旧的操作系统+利用MMX特性的处理器=所有一起工作。

SSE-SIMD寄存器扩展至128bits，独立于FPU。

AVX-另一种256bits扩展。

实际应用还包括内存复制(memcpy)和内存比较(memcmp)等等。

一个例子是：DES加密算法需要64-bits block，56-bits key,加密块生成64位结果。DES算法可以认为是一个非常大的电子电路，带有网格和AND/OR/NOT门。

Bitslice DES2—可以同时处理块和密钥。比如说unsigned int类型变量在X86下可以容纳32位，因此，使用64+56 unsigned int类型的变量，可以同时存储32个blocks-keys对。

我写了一个爆破Oracle RDBMS密码/哈希（基于DES）的工具。稍微修改了DES算法（SSE2和AVX）现在可以同时加密128或256block-keys对。

http://conus.info/utils/ops_SIMD/

25.1 Vectorization

向量化3，例如循环用两个数组生成一个数组。循环体从输入数组中取值，处理后存储到另一个数组。重要的一点是操作了每一个元素。向量化—同时处理多个元素。

向量化并不是新的技术：本书的作者在1998年使用Cray Y-MP EL“lite”时从Cray Y-MP supercomputer line看到过。

例子：

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

这段代码从A和B中取出元素，相乘，并把结果保存到C。

如果每个元素为32位int型，那么可以从A中加载4个元素到128bits XMM寄存器，B加载到另一个XMM寄存器，通过执行PMULID（Multiply Packed Signed Dword Integers and Store Low Result）和PMULHW(Multiply Packed Signed Integers and Store High Result)，一次可以得到4个64位结果。

循环次数从1024变成1024/4，当然更快。

25.1.1 Addition example

一些简单的情况下某些编译器可以自动向量化，Intel C++5.

函数如下：

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
        for (int i=0; i<sz; i++)
                ar3[i]=ar1[i]+ar2[i];
        return 0;
};
```

Intel C++

Intel C++ 11.1.051 win32下编译：

`icl intel.cpp /QaxSSE2 /Faintel.asm /Ox`

可以得到(IDA中):

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10          = dword ptr -10h
sz              = dword ptr 4
ar1             = dword ptr 8
ar2             = dword ptr 0Ch
```

```
ar3             = dword ptr 10h
                push    edi
                push    esi
                push    ebx
                push    esi
                mov     edx, [esp+10h+sz]
                test    edx, edx
                jle     loc_15B
                mov     eax, [esp+10h+ar3]
                cmp     edx, 6
                jle     loc_143
                cmp     eax, [esp+10h+ar2]
                jbe     short loc_36
                mov     esi, [esp+10h+ar2]
                sub     esi, eax
                lea     ecx, ds:0[edx*4]
                neg     esi
                cmp     ecx, esi
                jbe     short loc_55

loc_36:                                         ; CODE XREF: f(i
nt,int *,int *,int *)+21
                cmp     eax, [esp+10h+ar2]
                jnb     loc_143
                mov     esi, [esp+10h+ar2]
                sub     esi, eax
                lea     ecx, ds:0[edx*4]
                cmp     esi, ecx
                jb      loc_143
loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
                cmp     eax, [esp+10h+ar1]
                jbe     short loc_67
                mov     esi, [esp+10h+ar1]
                sub     esi, eax
                neg     esi
                cmp     ecx, esi
                jbe     short loc_7F

loc_67:                                          ; CODE XREF: f(
int,int *,int *,int *)+59
                cmp     eax, [esp+10h+ar1]
                jnb     loc_143
                mov     esi, [esp+10h+ar1]
                sub     esi, eax
                cmp     esi, ecx
                jb      loc_143

loc_7F:                                          ; CODE XREF: f(
int,int *,int *,int *)+65
                mov     edi, eax ; edi = ar1
                and     edi, 0Fh ; is ar1 16-byte aligned?
                jz      short loc_9A ; yes
                test    edi, 3
```

```
                jnz     loc_162
                neg     edi
                add     edi, 10h
                shr     edi, 2

loc_9A:                                          ; CODE XREF: f(
int,int *,int *,int *)+84
                lea     ecx, [edi+4]
                cmp     edx, ecx
                jl      loc_162
                mov     ecx, edx
                sub     ecx, edi
                and     ecx, 3
                neg     ecx
                add     ecx, edx
                test    edi, edi
                jbe     short loc_D6
                mov     ebx, [esp+10h+ar2]
                mov     [esp+10h+var_10], ecx
                mov     ecx, [esp+10h+ar1]
                xor     esi, esi

loc_C1:                                          ; CODE XREF: f(
int,int *,int *,int *)+CD
                mov     edx, [ecx+esi*4]
                add     edx, [ebx+esi*4]
                mov     [eax+esi*4], edx
                inc     esi
                cmp     esi, edi
                jb      short loc_C1
                mov     ecx, [esp+10h+var_10]
                mov     edx, [esp+10h+sz]

loc_D6:                                           ; CODE XREF: f
(int,int *,int *,int *)+B2
                mov     esi, [esp+10h+ar2]
                lea     esi, [esi+edi*4] ; is ar2+i*4 16-byte al
igned?
                test    esi, 0Fh
                jz      short loc_109 ; yes!
                mov     ebx, [esp+10h+ar1]
                mov     esi, [esp+10h+ar2]

loc_ED:                                           ; CODE XREF: f
(int,int *,int *,int *)+105
                movdqu  xmm1, xmmword ptr [ebx+edi*4]
                movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
is not 16-byte aligned, so load
                it to   xmm0
                paddd   xmm1, xmm0
                movdqa  xmmword ptr [eax+edi*4], xmm1
                add     edi, 4
                cmp     edi, ecx
```

```
                jb      short loc_ED
                jmp     short loc_127
; ----------------------------------------------------------------
-------------
loc_109:                                          ; CODE XREF: f
(int,int *,int *,int *)+E3
                mov     ebx, [esp+10h+ar1]
                mov     esi, [esp+10h+ar2]
loc_111:                                          ; CODE XREF: f
(int,int *,int *,int *)+125
                movdqu  xmm0, xmmword ptr [ebx+edi*4]
                paddd   xmm0, xmmword ptr [esi+edi*4]
                movdqa  xmmword ptr [eax+edi*4], xmm0
                add     edi, 4
                cmp     edi, ecx
                jb      short loc_111

loc_127:                                          ; CODE XREF: f
(int,int *,int *,int *)+107
                                                  ; f(int,int *,
int *,int *)+164
                cmp     ecx, edx
                jnb     short loc_15B
                mov     esi, [esp+10h+ar1]
                mov     edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
                mov     ebx, [esi+ecx*4]
                add     ebx, [edi+ecx*4]
                mov     [eax+ecx*4], ebx
                inc     ecx
                cmp     ecx, edx
                jb      short loc_133
                jmp     short loc_15B
; ----------------------------------------------------------------
-------------
loc_143:                                          ; CODE XREF: f
(int,int *,int *,int *)+17
                                                  ; f(int,int *,
int *,int *)+3A ...
                mov     esi, [esp+10h+ar1]
                mov     edi, [esp+10h+ar2]
                xor     ecx, ecx

loc_14D:                                          ; CODE XREF: f
(int,int *,int *,int *)+159
                mov     ebx, [esi+ecx*4]
                add     ebx, [edi+ecx*4]
                mov     [eax+ecx*4], ebx
                inc     ecx
                cmp     ecx, edx
                jb      short loc_14D
```

```
loc_15B:                                                  ; CODE XREF: f
(int,int *,int *,int *)+A
                                                          ; f(int,int *,
int *,int *)+129 ...
                xor     eax, eax
                pop     ecx
                pop     ebx
                pop     esi
                pop     edi
                retn
; ---------------------------------------------------------------------
-------------
loc_162:                                                   ; CODE XREF:
f(int,int *,int *,int *)+8C
                                                           ; f(int,int *
,int *,int *)+9F
                xor     ecx, ecx
                jmp     short loc_127
?f@@YAHHPAH00@Z endp
```

SSE2相关指令是：

```
MOVDQU (Move Unaligned Double Quadword)—仅仅从内存加载16个字节到XMM
寄存器。
PADDD (Add Packed Integers)—把源存储器与目的寄存器按双字对齐无符号整数
普通相加,结果送入目的寄存器,内存变量必须对齐内存16字节.
MOVDQA (Move Aligned Double Quadword)—把源存储器内容值送入目的寄存器,
当有m128时,必须对齐内存16字节.
```

如果工作元素超过4对，并且指针ar3按照16字节对齐，SSE2指令将被执行： 如果ar2按照16字节对齐，则代码如下：

```
movdqu  xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddd   xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa  xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

否则,ar2处的值将用MOVDQU加载到XMM0，它不需要对齐指针，代码如下：

```
movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte a
ligned, so load it to xmm0
paddd   xmm1, xmm0
movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

其他情况，将没有SSE2代码被执行。

GCC

gcc用-O3 选项同时打开SSE2支持: -msse2.

可以得到(GCC 4.4.1):

```
; f(int, int *, int *, int *)
                public _Z1fiPiS_S_
_Z1fiPiS_S_     proc near

var_18          = dword ptr -18h
var_14          = dword ptr -14h
var_10          = dword ptr -10h
arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch
arg_8           = dword ptr 10h
arg_C           = dword ptr 14h
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                sub     esp, 0Ch
                mov     ecx, [ebp+arg_0]
                mov     esi, [ebp+arg_4]
                mov     edi, [ebp+arg_8]
                mov     ebx, [ebp+arg_C]
                test    ecx, ecx
                jle     short loc_80484D8
                cmp     ecx, 6
                lea     eax, [ebx+10h]
                ja      short loc_80484E8

loc_80484C1:                            ; CODE XREF: f(int,int *,int *,i
nt *)+4B
                                        ; f(int,int *,int *,int *)+61 ..
.
                xor     eax, eax
                nop
                lea     esi, [esi+0]
loc_80484C8:                            ; CODE XREF: f(int,int *,int *,i
nt *)+36
                mov     edx, [edi+eax*4]
                add     edx, [esi+eax*4]
                mov     [ebx+eax*4], edx
                add     eax, 1
                cmp     eax, ecx
                jnz     short loc_80484C8

loc_80484D8:                            ; CODE XREF: f(int,int *,int *,i
nt *)+17
                                        ; f(int,int *,int *,int *)+A5
```

```
                add     esp, 0Ch
                xor     eax, eax
                pop     ebx
                pop     esi
                pop     edi
                pop     ebp
                retn
; ---------------------------------------------------------------
-------------
                align 8
loc_80484E8:                     ; CODE XREF: f(int,int *,int *,i
nt *)+1F
                test    bl, 0Fh
                jnz     short loc_80484C1
                lea     edx, [esi+10h]
                cmp     ebx, edx
                jbe     loc_8048578

loc_80484F8:                     ; CODE XREF: f(int,int *,int *,i
nt *)+E0
                lea     edx, [edi+10h]
                cmp     ebx, edx
                ja      short loc_8048503
                cmp     edi, eax
                jbe     short loc_80484C1

loc_8048503:                     ; CODE XREF: f(int,int *,int *,i
nt *)+5D
                mov     eax, ecx
                shr     eax, 2
                mov     [ebp+var_14], eax
                shl     eax, 2
                test    eax, eax
                mov     [ebp+var_10], eax
                jz      short loc_8048547
                mov     [ebp+var_18], ecx
                mov     ecx, [ebp+var_14]
                xor     eax, eax
                xor     edx, edx
                nop

loc_8048520:                     ; CODE XREF: f(int,int *,int *,i
nt *)+9B
                movdqu  xmm1, xmmword ptr [edi+eax]
                movdqu  xmm0, xmmword ptr [esi+eax]
                add     edx, 1
                paddd   xmm0, xmm1
                movdqa  xmmword ptr [ebx+eax], xmm0
                add     eax, 10h
                cmp     edx, ecx
                jb      short loc_8048520
                mov     ecx, [ebp+var_18]
                mov     eax, [ebp+var_10]
```

```
                cmp     ecx, eax
                jz      short loc_80484D8

loc_8048547:                    ; CODE XREF: f(int,int *,int *,i
nt *)+73
                lea     edx, ds:0[eax*4]
                add     esi, edx
                add     edi, edx
                add     ebx, edx
                lea     esi, [esi+0]

loc_8048558:                    ; CODE XREF: f(int,int *,int *,i
nt *)+CC
                mov     edx, [edi]
                add     eax, 1
                add     edi, 4
                add     edx, [esi]
                add     esi, 4
                mov     [ebx], edx
                add     ebx, 4
                cmp     ecx, eax
                jg      short loc_8048558
                add     esp, 0Ch
                xor     eax, eax
                pop     ebx
                pop     esi
                pop     edi
                pop     ebp
                retn
; ---------------------------------------------------------------
-------------
loc_8048578:                    ; CODE XREF: f(int,int *,int *,i
nt *)+52
                cmp     eax, esi
                jnb     loc_80484C1
                jmp     loc_80484F8
_Z1fiPiS_S_     endp
```

几乎一样，但没有Intel的细致。

25.1.2 Memory copy example

22.2 SIMD strlen() implementation

SIMD指令可能通过特殊的宏8插入到C/C++代码中。MSVC中他们被保存在intrin.h中。

Strlen()函数9的实现使用了SIMD指令，比常规的实现快了2-2.5倍。该函数将16个字符加载到一个XMM寄存器并检查是否为零

```
size_t strlen_sse2(const char *str)
{
        register size_t len = 0;
        const char *s=str;
        bool str_is_aligned=(((unsigned int)str)&0xFFFFFFF0) ==
(unsigned int)str;

        if (str_is_aligned==false)
                return strlen (str);

        __m128i xmm0 = _mm_setzero_si128();
        __m128i xmm1;
        int mask = 0;

        for (;;)
        {
                xmm1 = _mm_load_si128((__m128i *)s);
                xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
                if ((mask = _mm_movemask_epi8(xmm1)) != 0)
                {
                        unsigned long pos;
                        _BitScanForward(&pos, mask);
                        len += (size_t)pos;
                        break;
                }
                s += sizeof(__m128i);
                len += sizeof(__m128i);
        };

        return len;
}
```

(这里的例子基于源代码).

MSVC 2010 /Ox 编译选项:

```
_pos$75552 = -4                 ; size = 4
_str$ = 8                       ; size = 4
?strlen_sse2@@YAIPBD@Z PROC     ; strlen_sse2

    push    ebp
    mov     ebp, esp
    and     esp, -16 ; fffffff0H
    mov     eax, DWORD PTR _str$[ebp]
    sub     esp, 12 ; 0000000cH
    push    esi
    mov     esi, eax
    and     esi, -16 ; fffffff0H
    xor     edx, edx
    mov     ecx, eax
    cmp     esi, eax
```

```
    je      SHORT $LN4@strlen_sse
    lea     edx, DWORD PTR [eax+1]
    npad    3
$LL11@strlen_sse:
    mov     cl, BYTE PTR [eax]
    inc     eax
    test    cl, cl
    jne     SHORT $LL11@strlen_sse
    sub     eax, edx
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
$LN4@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [eax]
    pxor    xmm0, xmm0
    pcmpeqb         xmm1, xmm0
    pmovmskb        eax, xmm1
    test    eax, eax
    jne     SHORT $LN9@strlen_sse
$LL3@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [ecx+16]
    add     ecx, 16 ; 00000010H
    pcmpeqb         xmm1, xmm0
    add     edx, 16 ; 00000010H
    pmovmskb        eax, xmm1
    test    eax, eax
    je      SHORT $LL3@strlen_sse
$LN9@strlen_sse:
    bsf     eax, eax
    mov     ecx, eax
    mov     DWORD PTR _pos$75552[esp+16], eax
    lea     eax, DWORD PTR [ecx+edx]
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
?strlen_sse2@@YAIPBD@Z ENDP ; strlen_sse2
```

首先，检查str指针，如果不是按照16字节对齐则调用常规实现。

然后使用movdqa指令加载16个字节到xmm1.这里不使用movdqu的原因是如果指针不一致则从内存中加载的数据可能会不一致。

是的，它可能会以这种方式做，如果指针对齐，使用MOVDQA加载数据，否则使用比较慢的MOVDQU。

但是我们应该注意到这样的警告：

在windowsNT操作系统但不限于该操作系统，内存页按4kb对齐。每个win32进程独占4GB虚拟内存。事实上，只有部分地址空间与真实物理内存对应，如果进程访问的内存没有对应物理内存，将触发异常。这是虚拟内存的工作方式10.

一个函数一次加载16个字节，可能会跨内存分块访问。我们考虑这样一种情况，操作系统在x008c0000分配8192（0x2000）字节,因此块字节从地质0x008c0000到0x008c1fff。 内存块之后从0x008c2000什么都没有，操作系统没有分配任何内存。访问该地址将触发异常。

假如内存块包含的最后5个字符如下：

```
0x008c1ff8      'h'
0x008c1ff9      'e'
0x008c1ffa      'l'
0x008c1ffb      'l'
0x008c1ffc      'o'
0x008c1ffd      'x00'
0x008c1ffe      random noise
0x008c1fff      random noise
```

正常情况下，strlen()只会读取到"hello"。

如果我们使用MOVDQU读取16个字节，将会触发异常，应该避免这种情况。

因为我们要确保16字节对齐，保证我们不会读取未分配的内存。

让我们回到函数：

```
_mm_setzero_si128()—宏pxor xmm0, xmm0—清空XMM0寄存器。
_mm_load_si128()—宏 MOVDQA, 从内存加载16个字节到XMM寄存器。
_mm_cmpeq_epi8()—宏PCMPEQB,比较XMM寄存器的字节位，如果相等则为0xff否则
为0。
```

比如：

```
XMM1: 11223344556677880000000000000000
XMM0: 11ab3444007877881111111111111111
```

执行pcmpeqb xmm1, xmm0之后，XMM1寄存器的值为：

```
XMM1: ff0000ff0000ffff0000000000000000
```

在本例中该指令比较每一个16字节块与16字节0字节块对比，XMM0通过pxor xmm0 xmm0置零。

接下来宏_mm_movemask_epi8() —这是PMOVMSKB指令。

```
pmovmskb eax, xmm1
```

pmovmskb创建源操作数每一个字节的自高位掩码，并保存结果到目的操作数的低byte。源操作数必须为MMX寄存器，目的操作数必须为32位通用寄存器。

比如：

```
XMM1: 0000ff00000000000000ff0000000000
```

对应的EAX：

```
EAX=0010000000100000b
```

之后bsf eax,eax被执行，eax值为5，意味着第一个是1的位置是5（从0开始）。

MSVC关于这个指令的宏是：_BitScanForward.

至此，找到结尾0的位置，然后程序返回长度计数。

整个过程大致就是这样。

顺便提一下，MSVC为了优化，使用了两个并排的循环。

SSE 4.2(英特尔core i7)提供了更多的指令,这些可能更容易字符串操作。

http://www.strchr.com/strcmp_and_strlen_using_sse_4.2

第二十六章

64位化

26.1 x86-64

对x86架构来说这是一个64位的扩展。 从反编译工程师的角度来看，最重要的区别是： 几乎所有的寄存器（除了FPU和SIMD）都扩展到了64位，而且都有一个r-前缀，而且还额外添加了8个寄存器。 现在所有的通用寄存器是：RAX、RBX、RCX、RDX、RBP、RSP、RSI、RDI、R8、R9、R10、R11、R12、R13、R14、R15。 当然，还是可以像以前一样加载旧的寄存器的。比如，使用EAX就可以访问RAX的低32位部分。 新的R8-R15寄存器也有对应的低位：R8D-R15D（低32位）、R8W-R15W（低16位）、R8B-R15B（低8位）。 SIMD寄存器的数量从8个扩展到了16个：XMM0-XMM15。

在Win64下，函数调用转换有一些轻微的变化。例如fastcall（见47.3节）。最开始的4个参数将存储在RCX、RDX、R8和R9寄存器里，其他的保存在栈上。调用者函数必须分配32个字节，因此被调用者可以保存前4个参数，然后再去按照他自己的需要去利用这些寄存器。一些较短的函数可以直接从寄存器里使用参数，但是大点的函数就需要把参数保存到栈上了。 系统V AMD64 ABI（LINUX, *BSD, MAC OS X）也改变了fastcall的方式。它为前6个参数使用了6个寄存器RDI、RSI、RDX、RCX、R8、R9。剩余的参数将传入栈中。 请看调用转换（47）一节。

为了保证兼容性，C int类型依然是32位。 •现在所有的指针都是64位的了。 当然这个有时候很麻烦：因为现在我们需要2倍的空间来存储指针，包括缓存，而不管事实上64位CPU只会使用48位的扩展内存这个情况。

由于现在寄存器数量翻倍了，编译器也将有更多的空间来处理寄存器分配的策略。对我们来说，也就是现在提交的代码将会有更少的本地变量。 例如，DES加密算法中计算第一个S-Box时，使用位切割DES方法（见22章）他将每次处理32/64/128/256个变量（依据DES_type类型（uint32、uint64、SSE2或者AVX））。

```
/*
* Generated S-box files.
*
* This software may be modified, redistributed, and used for any
 purpose,
* so long as its origin is acknowledged.
*
* Produced by Matthew Kwan - March 1998
*/
#ifdef _WIN64
#define DES_type unsigned __int64
#else
```

```
#define DES_type unsigned int
#endif
void
s1 (
    DES_type a1,
    DES_type a2,
    DES_type a3,
    DES_type a4,
    DES_type a5,
    DES_type a6,
    DES_type *out1,
    DES_type *out2,
    DES_type *out3,
    DES_type *out4
) {
    DES_type x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type x49, x50, x51, x52, x53, x54, x55, x56;
    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x5;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
    *out2 ^= x22;
    x23 = x1 | x5;
    x24 = x23 ^ x8;
    x25 = x18 & ~x2;
    x26 = a2 & ~x25;
    x27 = x24 ^ x26;
    x28 = x6 | x7;
    x29 = x28 ^ x25;
    x30 = x9 ^ x24;
```

```
    x31 = x18 & ~x30;
    x32 = a2 & x31;
    x33 = x29 ^ x32;
    x34 = a1 & x33;
    x35 = x27 ^ x34;
    *out4 ^= x35;
    x36 = a3 & x28;
    x37 = x18 & ~x36;
    x38 = a2 | x3;
    x39 = x37 ^ x38;
    x40 = a3 | x31;
    x41 = x24 & ~x37;
    x42 = x41 | x3;
    x43 = x42 & ~a2;
    x44 = x40 ^ x43;
    x45 = a1 & ~x44;
    x46 = x39 ^ ~x45;
    *out1 ^= x46;
    x47 = x33 & ~x9;
    x48 = x47 ^ x39;
    x49 = x4 ^ x36;
    x50 = x49 & ~x5;
    x51 = x42 | x18;
    x52 = x51 ^ a5;
    x53 = a2 & ~x52;
    x54 = x50 ^ x53;
    x55 = a1 | x54;
    x56 = x48 ^ ~x55;
    *out3 ^= x56;
}
```

这儿也有许多本地变量。当然，并不是所有的这些都存在本地栈上。让我们用MSVC2008的/Ox选项来编译一下：

清单23.1 使用MSVC 2008编译

```
PUBLIC _s1
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_x6$ = -20 ; size = 4
_x3$ = -16 ; size = 4
_x1$ = -12 ; size = 4
_x8$ = -8 ; size = 4
_x4$ = -4 ; size = 4
_a1$ = 8 ; size = 4
_a2$ = 12 ; size = 4
_a3$ = 16 ; size = 4
_x33$ = 20 ; size = 4
_x7$ = 20 ; size = 4
_a4$ = 20 ; size = 4
_a5$ = 24 ; size = 4
```

```
tv326 = 28 ; size = 4
_x36$ = 28 ; size = 4
_x28$ = 28 ; size = 4
_a6$ = 28 ; size = 4
_out1$ = 32 ; size = 4
_x24$ = 36 ; size = 4
_out2$ = 36 ; size = 4
_out3$ = 40 ; size = 4
_out4$ = 44 ; size = 4
_s1 PROC
    sub esp, 20 ; 00000014H
    mov edx, DWORD PTR _a5$[esp+16]
    push ebx
    mov ebx, DWORD PTR _a4$[esp+20]
    push ebp
    push esi
    mov esi, DWORD PTR _a3$[esp+28]
    push edi
    mov edi, ebx
    not edi
    mov ebp, edi
    and edi, DWORD PTR _a5$[esp+32]
    mov ecx, edx
    not ecx
    and ebp, esi
    mov eax, ecx
    and eax, esi
    and ecx, ebx
    mov DWORD PTR _x1$[esp+36], eax
    xor eax, ebx
    mov esi, ebp
    or esi, edx
    mov DWORD PTR _x4$[esp+36], esi
    and esi, DWORD PTR _a6$[esp+32]
    mov DWORD PTR _x7$[esp+32], ecx
    mov edx, esi
    xor edx, eax
    mov DWORD PTR _x6$[esp+36], edx
    mov edx, DWORD PTR _a3$[esp+32]
    xor edx, ebx
    mov ebx, esi
    xor ebx, DWORD PTR _a5$[esp+32]
    mov DWORD PTR _x8$[esp+36], edx
    and ebx, edx
    mov ecx, edx
    mov edx, ebx
    xor edx, ebp
    or edx, DWORD PTR _a6$[esp+32]
    not ecx
    and ecx, DWORD PTR _a6$[esp+32]
    xor edx, edi
    mov edi, edx
    or edi, DWORD PTR _a2$[esp+32]
```

```
        mov DWORD PTR _x3$[esp+36], ebp
        mov ebp, DWORD PTR _a2$[esp+32]
        xor edi, ebx
        and edi, DWORD PTR _a1$[esp+32]
        mov ebx, ecx
        xor ebx, DWORD PTR _x7$[esp+32]
        not edi
        or ebx, ebp
        xor edi, ebx
        mov ebx, edi
        mov edi, DWORD PTR _out2$[esp+32]
        xor ebx, DWORD PTR [edi]
        not eax
        xor ebx, DWORD PTR _x6$[esp+36]
        and eax, edx
        mov DWORD PTR [edi], ebx
        mov ebx, DWORD PTR _x7$[esp+32]
        or ebx, DWORD PTR _x6$[esp+36]
        mov edi, esi
        or edi, DWORD PTR _x1$[esp+36]
        mov DWORD PTR _x28$[esp+32], ebx
        xor edi, DWORD PTR _x8$[esp+36]
        mov DWORD PTR _x24$[esp+32], edi
        xor edi, ecx
        not edi
        and edi, edx
        mov ebx, edi
        and ebx, ebp
        xor ebx, DWORD PTR _x28$[esp+32]
        xor ebx, eax
        not eax
        mov DWORD PTR _x33$[esp+32], ebx
        and ebx, DWORD PTR _a1$[esp+32]
        and eax, ebp
        xor eax, ebx
        mov ebx, DWORD PTR _out4$[esp+32]
        xor eax, DWORD PTR [ebx]
        xor eax, DWORD PTR _x24$[esp+32]
        mov DWORD PTR [ebx], eax
        mov eax, DWORD PTR _x28$[esp+32]
        and eax, DWORD PTR _a3$[esp+32]
        mov ebx, DWORD PTR _x3$[esp+36]
        or edi, DWORD PTR _a3$[esp+32]
        mov DWORD PTR _x36$[esp+32], eax
        not eax
        and eax, edx
        or ebx, ebp
        xor ebx, eax
        not eax
        and eax, DWORD PTR _x24$[esp+32]
        not ebp
        or eax, DWORD PTR _x3$[esp+36]
        not esi
```

```
    and ebp, eax
    or eax, edx
    xor eax, DWORD PTR _a5$[esp+32]
    mov edx, DWORD PTR _x36$[esp+32]
    xor edx, DWORD PTR _x4$[esp+36]
    xor ebp, edi
    mov edi, DWORD PTR _out1$[esp+32]
    not eax
    and eax, DWORD PTR _a2$[esp+32]
    not ebp
    and ebp, DWORD PTR _a1$[esp+32]
    and edx, esi
    xor eax, edx
    or eax, DWORD PTR _a1$[esp+32]
    not ebp
    xor ebp, DWORD PTR [edi]
    not ecx
    and ecx, DWORD PTR _x33$[esp+32]
    xor ebp, ebx
    not eax
    mov DWORD PTR [edi], ebp
    xor eax, ecx
    mov ecx, DWORD PTR _out3$[esp+32]
    xor eax, DWORD PTR [ecx]
    pop edi
    pop esi
    xor eax, ebx
    pop ebp
    mov DWORD PTR [ecx], eax
    pop ebx
    add esp, 20 ; 00000014H
    ret 0
_s1 ENDP
```

编译器在本地栈上分配了5个变量。 现在再让我们在MSVC 2008的64位环境中试一试：

清单23.2 使用MSVC 2008编译

```
a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1 PROC
```

```
$LN3:
mov QWORD PTR [rsp+24], rbx
mov QWORD PTR [rsp+32], rbp
mov QWORD PTR [rsp+16], rdx
mov QWORD PTR [rsp+8], rcx
push rsi
push rdi
push r12
push r13
push r14
push r15
mov r15, QWORD PTR a5$[rsp]
mov rcx, QWORD PTR a6$[rsp]
mov rbp, r8
mov r10, r9
mov rax, r15
mov rdx, rbp
not rax
xor rdx, r9
not r10
mov r11, rax
and rax, r9
mov rsi, r10
mov QWORD PTR x36$1$[rsp], rax
and r11, r8
and rsi, r8
and r10, r15
mov r13, rdx
mov rbx, r11
xor rbx, r9
mov r9, QWORD PTR a2$[rsp]
mov r12, rsi
or r12, r15
not r13
and r13, rcx
mov r14, r12
and r14, rcx
mov rax, r14
mov r8, r14
xor r8, rbx
xor rax, r15
not rbx
and rax, rdx
mov rdi, rax
xor rdi, rsi
or rdi, rcx
xor rdi, r10
and rbx, rdi
mov rcx, rdi
or rcx, r9
xor rcx, rax
mov rax, r13
xor rax, QWORD PTR x36$1$[rsp]
```

```
    and rcx, QWORD PTR a1$[rsp]
    or rax, r9
    not rcx
    xor rcx, rax
    mov rax, QWORD PTR out2$[rsp]
    xor rcx, QWORD PTR [rax]
    xor rcx, r8
    mov QWORD PTR [rax], rcx
    mov rax, QWORD PTR x36$1$[rsp]
    mov rcx, r14
    or rax, r8
    or rcx, r11
    mov r11, r9
    xor rcx, rdx
    mov QWORD PTR x36$1$[rsp], rax
    mov r8, rsi
    mov rdx, rcx
    xor rdx, r13
    not rdx
    and rdx, rdi
    mov r10, rdx
    and r10, r9
    xor r10, rax
    xor r10, rbx
    not rbx
    and rbx, r9
    mov rax, r10
    and rax, QWORD PTR a1$[rsp]
    xor rbx, rax
    mov rax, QWORD PTR out4$[rsp]
    xor rbx, QWORD PTR [rax]
    xor rbx, rcx
    mov QWORD PTR [rax], rbx
    mov rbx, QWORD PTR x36$1$[rsp]
    and rbx, rbp
    mov r9, rbx
    not r9
    and r9, rdi
    or r8, r11
    mov rax, QWORD PTR out1$[rsp]
    xor r8, r9
    not r9
    and r9, rcx
    or rdx, rbp
    mov rbp, QWORD PTR [rsp+80]
    or r9, rsi
    xor rbx, r12
    mov rcx, r11
    not rcx
    not r14
    not r13
    and rcx, r9
    or r9, rdi
```

```
    and rbx, r14
    xor r9, r15
    xor rcx, rdx
    mov rdx, QWORD PTR a1$[rsp]
    not r9
    not rcx
    and r13, r10
    and r9, r11
    and rcx, rdx
    xor r9, rbx
    mov rbx, QWORD PTR [rsp+72]
    not rcx
    xor rcx, QWORD PTR [rax]
    or r9, rdx
    not r9
    xor rcx, r8
    mov QWORD PTR [rax], rcx
    mov rax, QWORD PTR out3$[rsp]
    xor r9, r13
    xor r9, QWORD PTR [rax]
    xor r9, r8
    mov QWORD PTR [rax], r9
    pop r15
    pop r14
    pop r13
    pop r12
    pop rdi
    pop rsi
    ret 0
s1 ENDP
```

编译器在栈上并没有分配任何内存空间，x36是a5的同义词。 顺带一提，我们可以在这儿看到的是，函数在调用者空间中保存了RCX和RDX，但是R8和R9虽然在一开始就使用了，但是却并没有保存。 还有，还有拥有更多GPR的CPU，比如Itanium（有128个寄存器）。

26.2 ARM

在ARM中，64位指令在ARMv8中才开始出现。

26.3 浮点数字

见24章以了解更多的x86-64处理器中是如何处理浮点数的。

第二十七章

使用SIMD来处理浮点数

当然，在增加了x64扩展这个特性之后，FPU在x86兼容处理器中还是存在的。但是同事，SIMD扩展（SSE, SSE2等）已经有了，他们也可以处理浮点数。数字格式依然相同（使用IEEE754标准）。

所以，x86-64编译器通常都使用SIMD指令。可以说这是一个好消息，因为这让我们可以更容易的使用他们。

24.1 简单的例子

```
double f (double a, double b)
{
    return a/3.14 + b*4.1;
};
```

27.1.1 x64

清单24.1： MSFC 2012 x64 /Ox

```
__real@4010666666666666 DQ 04010666666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14
a$ = 8
b$ = 16
f PROC
    divsd xmm0, QWORD PTR __real@40091eb851eb851f
    mulsd xmm1, QWORD PTR __real@4010666666666666
    addsd xmm0, xmm1
    ret 0
f ENDP
```

输入的浮点数被传入了XMM0-XMM3寄存器，其他的通过栈来传递。 a被传入了XMM0，b则是通过XMM1。 XMM寄存器是128位的（可以参考SIMD22一节），但是我们的类型是double型的，也就意味着只有一半的寄存器会被使用。

DIVSD是一个SSE指令，意思是“Divide Scalar Double-Precision Floating-Point Values”（除以标量双精度浮点数值），它只是把一个double除以另一个double，然后把结果存在操作符的低一半位中。 常量会被编译器以IEEE754格式提前编码。 MULSD和ADDSD也是类似的，只不过一个是乘法，一个是加法。 函数处理double的结果将保存在XMM0寄存器中。

这是无优化的MSVC编译器的结果：

清单24.2： MSVC 2012 x64

```
__real@4010666666666666 DQ 04010666666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14
a$ = 8
b$ = 16
f PROC
    movsdx QWORD PTR [rsp+16], xmm1
    movsdx QWORD PTR [rsp+8], xmm0
    movsdx xmm0, QWORD PTR a$[rsp]
    divsd xmm0, QWORD PTR __real@40091eb851eb851f
    movsdx xmm1, QWORD PTR b$[rsp]
    mulsd xmm1, QWORD PTR __real@4010666666666666
    addsd xmm0, xmm1
    ret 0
f ENDP
```

有一些繁杂，输入参数保存在“shadow space”（影子空间，7.2.1节），但是只有低一半的寄存器，也即只有64位存了这个double的值。

27.1.2 x86

GCC编译器生成了几乎一样的代码。

24.2 通过参数传递浮点型变量

```
#include <math.h>
#include <stdio.h>
int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));
    return 0;
}
```

他们通过XMM0-XMM3的低一半寄存器传递。

清单24.3： MSVC 2012 x64 /Ox

```
$SG1354 DB '32.01 ^ 1.54 = %lf', 0aH, 00H
__real@40400147ae147ae1 DQ 040400147ae147ae1r ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r ; 1.54
main PROC
    sub rsp, 40 ; 00000028H
    movsdx xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
    movsdx xmm0, QWORD PTR __real@40400147ae147ae1
    call pow
    lea rcx, OFFSET FLAT:$SG1354
    movaps xmm1, xmm0
    movd rdx, xmm1
    call printf
    xor eax, eax
    add rsp, 40 ; 00000028H
    ret 0
main ENDP
```

在Intel和AMD的手册中（见14章和1章）并没有MOVSDX这个指令，而只有MOVSD一个。所以在x86中有两个指令共享了同一个名字（另一个见B.6.2）。显然，微软的开发者想要避免弄得一团糟，所以他们把它重命名为MOVSDX，它只是会多把一个值载入XMM寄存器的低一半中。 pow（）函数从XMM0和XMM1中加载参数，然后返回结果到XMM0中。 然后把值移动到RDX中，因为接下来printf()需要调用这个函数。为什么？老实说我也不知道，也许是因为printf()是一个参数不定的函数？

清单24.4：GCC 4.4.6 x64 -O3

```
.LC2:
.string "32.01 ^ 1.54 = %lf\n"
main:
    sub rsp, 8
    movsd xmm1, QWORD PTR .LC0[rip]
    movsd xmm0, QWORD PTR .LC1[rip]
    call pow
    ; result is now in XMM0
    mov edi, OFFSET FLAT:.LC2
    mov eax, 1 ; number of vector registers passed
    call printf
    xor eax, eax
    add rsp, 8
    ret
.LC0:
    .long 171798692
    .long 1073259479
.LC1:
    .long 2920577761
    .long 1077936455
```

GCC让结果更清晰，printf（）的值传入到了XMM0中。顺带一提，这是一个因为printf()才把1写入EAX中的例子。这意味着参数会被传递到向量寄存器中，就像标准需求一样（见21章）。

27.3 比较式的例子

```
double d_max (double a, double b)
{
    if (a>b)
    return a;
    return b;
};
```

27.3.1 x64

清单 24.5： MSVC 2012 x64 /Ox

```
a$ = 8
b$ = 16
d_max PROC
    comisd xmm0, xmm1
    ja SHORT $LN2@d_max
    movaps xmm0, xmm1
$LN2@d_max:
    fatret 0
d_max ENDP
```

优化过的MSVC产生了很容易理解的代码。 COMISD是“Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS”（比较标量双精度浮点数的值然后设置EFLAG）的缩写，显然，看着名字就知道他要干啥了。 非优化的MSVC代码产生了更加丰富的代码，但是仍然不难理解：

清单 24.6： MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max PROC
    comisd xmm0, xmm1
    ja SHORT $LN2@d_max
    movaps xmm0, xmm1
    $LN2@d_max:
    fatret 0
d_max ENDP
```

但是，GCC 4.4.6生成了更多的优化代码，并且使用了MAXSD（“Return Maximum Scalar Double-Precision Floating-Point Value”，返回最大的双精度浮点数的值）指令，它将选中其中一个最大数。

清单24.7： GCC 4.4.6 x64 -O3

```
a$ = 8
b$ = 16
d_max PROC
    movsdx QWORD PTR [rsp+16], xmm1
    movsdx QWORD PTR [rsp+8], xmm0
    movsdx xmm0, QWORD PTR a$[rsp]
    comisd xmm0, QWORD PTR b$[rsp]
    jbe SHORT $LN1@d_max
    movsdx xmm0, QWORD PTR a$[rsp]
    jmp SHORT $LN2@d_max
    $LN1@d_max:
    movsdx xmm0, QWORD PTR b$[rsp]
    $LN2@d_max:
    fatret 0
d_max ENDP
```

27.3.2 x86

27.4 Calculating machine epsilon: x64 and SIMD

27.5 回顾伪随机书生成器

27.6 总结

只有低一半的XMM寄存器会被使用，一组IEEE754格式的数字也会被存在这里。显然，所有的指令都有SD后缀（标量双精度数），这些操作数是可以用于IEEE754浮点数的，他们存在XMM寄存器的低64位中。 比FPU更简单的是，显然SIMD扩展并不像FPU以前那么混乱，栈寄存器模型也没使用。 如果你像试着将例子中的double替换成float的话，它们还是会使用同样的指令，但是后缀是SS（标量单精度数），例如MOVSS，COMISS，ADDSS等等。 标量（Scalar）代表着SIMD寄存器会包含仅仅一个值，而不是所有的。可以在所有类型的值中生效的指令都被“封装”成同一个名字。

第二十八章

关于ARM的具体细节

第二十九章

关于MIPS的具体细节

Part II 重要的基础知识

第三十章

有符号数的表示

第三十一章

字节顺序

第三十二章

内存

第三十三章

CPU

第三十四章

哈希函数

Part III 更高级些的例子

温度转换

另一个在初学者的编程书中常见的例子是温度转换程序，例如将华氏度转为摄氏度，或者反过来。

我也添加了一个简单的错误处理： 1）我们应该检查用户是否输入了正确的数字 2）我们应该检查摄氏度是否低于-273 °C，因为这比绝对零度还低，学校物理课上的东西应该都还记得。 exit()函数将立即终止程序，而不会回到调用者函数。

35.1 整数值

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int celsius, fahr;
    printf ("Enter temperature in Fahrenheit:\n");
    if (scanf ("%d", &fahr)!=1)
    {
        printf ("Error while parsing your input\n");
        exit(0);
    };
    celsius = 5 * (fahr-32) / 9;
    if (celsius<-273)
    {
        printf ("Error: incorrect temperature!\n");
        exit(0);
    };
    printf ("Celsius: %d\n", celsius);
};
```

35.1.1 MSVC 2012 x86

清单35.1： MSVC 2012 x86

```
$SG4228 DB ’Enter temperature in Fahrenheit:’, 0aH, 00H
$SG4230 DB ’%d’, 00H
$SG4231 DB ’Error while parsing your input’, 0aH, 00H
$SG4233 DB ’Error: incorrect temperature!’, 0aH, 00H
$SG4234 DB ’Celsius: %d’, 0aH, 00H
_fahr$ = -4 ; size = 4
_main PROC
    push ecx
    push esi
```

```
    mov esi, DWORD PTR __imp__printf
    push OFFSET $SG4228 ; 'Enter temperature in Fahrenheit:'
    call esi ; call printf()
    lea eax, DWORD PTR _fahr$[esp+12]
    push eax
    push OFFSET $SG4230 ; '%d'
    call DWORD PTR __imp__scanf
    add esp, 12 ; 0000000cH
    cmp eax, 1
    je SHORT $LN2@main
    push OFFSET $SG4231 ; 'Error while parsing your input'
    call esi ; call printf()
    add esp, 4
    push 0
    call DWORD PTR __imp__exit
    $LN9@main:
    $LN2@main:
    mov eax, DWORD PTR _fahr$[esp+8]
    add eax, -32 ; ffffffe0H
    lea ecx, DWORD PTR [eax+eax*4]
    mov eax, 954437177 ; 38e38e39H
    imul ecx
    sar edx, 1
    mov eax, edx
    shr eax, 31 ; 0000001fH
    add eax, edx
    cmp eax, -273 ; fffffeefH
    jge SHORT $LN1@main
    push OFFSET $SG4233 ; 'Error: incorrect temperature!'
    call esi ; call printf()
    add esp, 4
    push 0
    call DWORD PTR __imp__exit
    $LN10@main:
    $LN1@main:
    push eax
    push OFFSET $SG4234 ; 'Celsius: %d'
    call esi ; call printf()
    add esp, 8
    ; return 0 - at least by C99 standard
    xor eax, eax
    pop esi
    pop ecx
    ret 0
$LN8@main:
_main ENDP
```

关于这个我们可以说的是：

- printf()的地址先被载入了ESI寄存器中，所以printf()调用的序列会被CALL ESI处理，这是一个非常著名的编译器技术，当代码中存在多个序列调用同一个函数的时候，并且/或者有空闲的寄存器可以用上的时候，编译器就会这么做。

- 我们知道ADD EAX,-32指令会把EAX中的数据减去32。 EAX = EAX + (-32)等同于 EAX = EAX - 32，因此编译器决定用ADD而不是用SUB，也许这样性能比较高吧。
- LEA指令在值应当乘以5的时候用到了： lea ecx, DWORD PTR [eax+eax*4]*。是的，*i + i* 4是等同于i*5的，而且LEA比IMUL运行的要快。 还有，SHL EAX,2/ ADD EAX,EAX指令对也可以替换这句，而且有些编译器就是会这么优化。
- 用乘法做除法的技巧也会在这儿用上。
- 虽然我们没有指定，但是main()函数依然会返回0。C99规范告诉我们[15章，5.1.2.2.3] main()将在没有return时也会照常返回0。 这个规则仅仅对main()函数有效。 虽然MSVC并不支持C99，但是这么看说不好他还是做到了一部分呢？

35.1.2 MSVC 2012 x64 /Ox

生成的代码几乎一样，但是我发现每个exit()调用之后都有INT 3。

```
xor ecx, ecx
call QWORD PTR __imp_exit
int 3
```

INT 3是一个调试器断点。 可以知道的是exit()是永远不会return的函数之一。所以如果他“返回”了，那么估计发生了什么奇怪的事情，也是时候启动调试器了。

35.2 浮点数值

清单35.1: MSVC 2010

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    double celsius, fahr;
    printf ("Enter temperature in Fahrenheit:\n");
    if (scanf ("%lf", &fahr)!=1)
    {
        printf ("Error while parsing your input\n");
        exit(0);
    };
    celsius = 5 * (fahr-32) / 9;
    if (celsius<-273)
    {
        printf ("Error: incorrect temperature!\n");
        exit(0);
    };
    printf ("Celsius: %lf\n", celsius);
};
```

MSVC 2010 x86使用FPU指令...

清单35.2: MSVC 2010 x86 /Ox

```
$SG4038 DB 'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4040 DB '%lf', 00H
$SG4041 DB 'Error while parsing your input', 0aH, 00H
$SG4043 DB 'Error: incorrect temperature!', 0aH, 00H
$SG4044 DB 'Celsius: %lf', 0aH, 00H
__real@c071100000000000 DQ 0c071100000000000r ; -273
__real@4022000000000000 DQ 04022000000000000r ; 9
__real@4014000000000000 DQ 04014000000000000r ; 5
__real@4040000000000000 DQ 04040000000000000r ; 32
_fahr$ = -8 ; size = 8
_main PROC
    sub esp, 8
    push esi
    mov esi, DWORD PTR __imp__printf
    push OFFSET $SG4038 ; 'Enter temperature in Fahrenheit:'
    call esi ; call printf
    lea eax, DWORD PTR _fahr$[esp+16]
    push eax
    push OFFSET $SG4040 ; '%lf'
    call DWORD PTR __imp__scanf
    add esp, 12 ; 0000000cH
    cmp eax, 1
    je SHORT $LN2@main
    push OFFSET $SG4041 ; 'Error while parsing your input'
    call esi ; call printf
    add esp, 4
    push 0
```

```
    call DWORD PTR __imp__exit
    $LN2@main:
    fld QWORD PTR _fahr$[esp+12]
    fsub QWORD PTR __real@4040000000000000 ; 32
    fmul QWORD PTR __real@4014000000000000 ; 5
    fdiv QWORD PTR __real@4022000000000000 ; 9
    fld QWORD PTR __real@c071100000000000 ; -273
    fcomp ST(1)
    fnstsw ax
    test ah, 65 ; 00000041H
    jne SHORT $LN1@main
    push OFFSET $SG4043 ; 'Error: incorrect temperature!'
    fstp ST(0)
    call esi ; call printf
    add esp, 4
    push 0
    call DWORD PTR __imp__exit
    $LN1@main:
    sub esp, 8
    fstp QWORD PTR [esp]
    push OFFSET $SG4044 ; 'Celsius: %lf'
    call esi
    add esp, 12 ; 0000000cH
    ; return 0
    xor eax, eax
    pop esi
    add esp, 8
    ret 0
$LN10@main:
_main ENDP
```

但是MSVC从2012年开始又改成了使用SIMD指令：

清单35.3: MSVC 2010 x86 /Ox

```
$SG4228 DB 'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB '%lf', 00H
$SG4231 DB 'Error while parsing your input', 0aH, 00H
$SG4233 DB 'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB 'Celsius: %lf', 0aH, 00H
__real@c071100000000000 DQ 0c071100000000000r ; -273
__real@4040000000000000 DQ 04040000000000000r ; 32
__real@4022000000000000 DQ 04022000000000000r ; 9
__real@4014000000000000 DQ 04014000000000000r ; 5
_fahr$ = -8 ; size = 8
_main PROC
    sub esp, 8
    push esi
    mov esi, DWORD PTR __imp__printf
    push OFFSET $SG4228 ; 'Enter temperature in Fahrenheit:'
    call esi ; call printf
```

```
    lea eax, DWORD PTR _fahr$[esp+16]
    push eax
    push OFFSET $SG4230 ; '%lf'
    call DWORD PTR __imp__scanf
    add esp, 12 ; 0000000cH
    cmp eax, 1
    je SHORT $LN2@main
    push OFFSET $SG4231 ; 'Error while parsing your input'
    call esi ; call printf
    add esp, 4
    push 0
    call DWORD PTR __imp__exit
    $LN9@main:
    $LN2@main:
    movsd xmm1, QWORD PTR _fahr$[esp+12]
    subsd xmm1, QWORD PTR __real@4040000000000000 ; 32
    movsd xmm0, QWORD PTR __real@c071100000000000 ; -273
    mulsd xmm1, QWORD PTR __real@4014000000000000 ; 5
    divsd xmm1, QWORD PTR __real@4022000000000000 ; 9
    comisd xmm0, xmm1
    jbe SHORT $LN1@main
    push OFFSET $SG4233 ; 'Error: incorrect temperature!'
    call esi ; call printf
    add esp, 4
    push 0
    call DWORD PTR __imp__exit
    $LN10@main:
    $LN1@main:
    sub esp, 8
    movsd QWORD PTR [esp], xmm1
    push OFFSET $SG4234 ; 'Celsius: %lf'
    call esi ; call printf
    add esp, 12 ; 0000000cH
    ; return 0
    xor eax, eax
    pop esi
    add esp, 8
    ret 0
$LN8@main:
_main ENDP
```

当然，SIMD在x86下也是可用的，包括这些浮点数的运算。使用他们计算起来也确实方便点，所以微软编译器使用了他们。 我们也可以注意到 -273 这个值会很早的被载入XMM0。这个没问题，因为编译器并不一定会按照源代码里面的顺序产生代码。

第三十六章

斐波那契数列

另一个在编程教材中普遍使用的例子是，一个用来生成斐波那契数列的递归函数。

这个序列非常简单：每个数字都是前面两个数字的和。打头的两个数字都是1或者是0,1,1。

该序列起始是这样的：

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181...

36.1 例一

这个实现起来比较简单。下面这个程序产生直到21的序列。

```
#include <stdio.h>
void fib (int a, int b, int limit)
{
    printf ("%d\n", a+b);
    if (a+b > limit)
        return;
    fib (b, a+b, limit);
};

int main()
{
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
};
```

Listing 36.1: MSVC 2010 x86

```
_a$ = 8          ; size = 4
_b$ = 12         ; size = 4
_limit$ = 16     ; size = 4
_fib    PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    push    eax
    push    OFFSET $SG2643
    call    DWORD PTR __imp__printf
    add     esp, 8
    mov     ecx, DWORD PTR _a$[ebp]
    add     ecx, DWORD PTR _b$[ebp]
    cmp     ecx, DWORD PTR _limit$[ebp]
    jle     SHORT $LN1@fib
    jmp     SHORT $LN2@fib
$LN1@fib:
    mov     edx, DWORD PTR _limit$[ebp]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    call    _fib
    add     esp, 12
$LN2@fib:
    pop     ebp
    ret     0
_fib    ENDP

_main   PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2647
    call    DWORD PTR __imp__printf
    add     esp, 4
    push    20
    push    1
    push    1
    call    _fib
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0
_main   ENDP
```

我们将用这个来说明一下栈帧。

让我们在OllyDbg中加载这个例子，并且跟踪到最后一次对 `f()` 函数的调用：

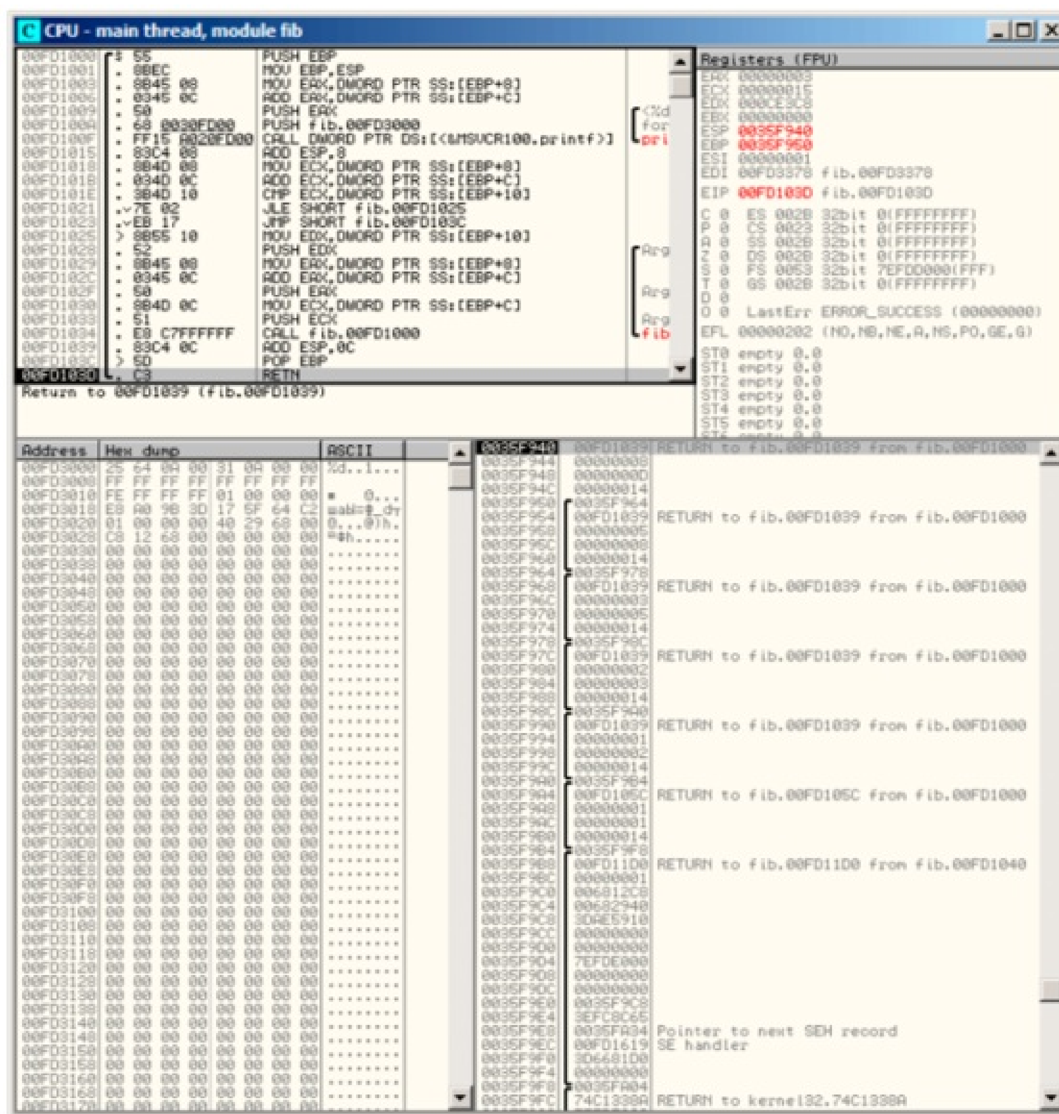


图 36.1: OllyDbg: 最后一次对 `f()` 的调用

让我们来更加仔细地研究一下栈。本书的作者向其中加了一些注释（在这个例子中，就是把OllyDbg中的多个条目copy到剪切板中(Ctrl-C)）：

```
0035F940  00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F944  00000008 1st argument: a
0035F948  0000000D 2nd argument: b
0035F94C  00000014 3rd argument: limit
0035F950 /0035F964 saved EBP register
0035F954 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F958 |00000005 1st argument: a
0035F95C |00000008 2nd argument: b
0035F960 |00000014 3rd argument: limit
0035F964 ]0035F978 saved EBP register
0035F968 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F96C |00000003 1st argument: a
0035F970 |00000005 2nd argument: b
0035F974 |00000014 3rd argument: limit
0035F978 ]0035F98C saved EBP register
0035F97C |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F980 |00000002 1st argument: a
0035F984 |00000003 2nd argument: b
0035F988 |00000014 3rd argument: limit
0035F98C ]0035F9A0 saved EBP register
0035F990 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F994 |00000001 1st argument: a
0035F998 |00000002 2nd argument: b
0035F99C |00000014 3rd argument: limit
0035F9A0 ]0035F9B4 saved EBP register
0035F9A4 |00FD105C RETURN to fib.00FD105C from fib.00FD1000
0035F9A8 |00000001 1st argument: a           \
0035F9AC |00000001 2nd argument: b           | prepared in main(
) for f1()
0035F9B0 |00000014 3rd argument: limit       /
0035F9B4 ]0035F9F8 saved EBP register
0035F9B8 |00FD11D0 RETURN to fib.00FD11D0 from fib.00FD1040
0035F9BC |00000001 main() 1st argument: argc \
0035F9C0 |006812C8 main() 2nd argument: argv | prepared in CRT f
or main()
0035F9C4 |00682940 main() 3rd argument: envp /
```

该函数是递归的，因此看起来就像个“三明治”。我们能够看出参数limit总是相同的（0x14或20），但是参数a和b在每次调用时都是不同的。其中也有RA（Return Address，返回地址）和保存的EBP值。OllyDbg可以决定基于EBP的帧，所以就画出了这些中括号（]）。每个中括号中的值构成了栈帧，换句话说，每一个函数都使用栈来作为暂存空间。我们也可以说每一个函数都不能访问超出其帧边界的栈元素（不包括函数参数），虽然这在技术上是有可能的。上一句话通常是正确的，除非函数中有了bug。每个保存的EBP值为前一栈帧的地址：这就是有些调试器可以很容易地划分在帧中的栈和dump每个函数参数的原因。

正如我们在这里所见，每一个函数都为下一个函数调用准备好了参数。

在最后有用于 `main()` 函数的三个参数。argc值为1（是的，我们确实没有用命令行参数来运行程序）。

这样很容易导致栈溢出：只是删除（或注释）掉limit检测，程序就会抛出0xC00000FD异常而崩溃（stack overflow）。

36.2 例二

我构造的函数有些冗余，所以就让我们来添加一个局部变量next并用它代替所有的"a+b"：

```
#include <stdio.h>
void fib (int a, int b, int limit)
{
    int next=a+b;
    printf ("%d\n", next);
    if (next > limit)
        return;
    fib (b, next, limit);
};

int main()
{
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
};
```

以下的输出是MSVC非优化编译的输出，所以next变量在局部栈中分配空间。

```
_next$ = -4     ; size = 4
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_limit$ = 16    ; size = 4
_fib    PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    mov     DWORD PTR _next$[ebp], eax
    mov     ecx, DWORD PTR _next$[ebp]
    push    ecx
    push    OFFSET $SG2751 ; '%d'
    call    DWORD PTR __imp__printf
    add     esp, 8
    mov     edx, DWORD PTR _next$[ebp]
    cmp     edx, DWORD PTR _limit$[ebp]
    jle     SHORT $LN1@fib
    jmp     SHORT $LN2@fib
$LN1@fib:
    mov     eax, DWORD PTR _limit$[ebp]
    push    eax
    mov     ecx, DWORD PTR _next$[ebp]
    push    ecx
    mov     edx, DWORD PTR _b$[ebp]
    push    edx
    call    _fib
    add     esp, 12
$LN2@fib:
    mov     esp, ebp
    pop     ebp
    ret     0
_fib    ENDP

_main   PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2753 ; "0\n1\n1\n"
    call    DWORD PTR __imp__printf
    add     esp, 4
    push    20
    push    1
    push    1
    call    _fib
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0
_main    ENDP
```

让我再一次加载OllyDbg：

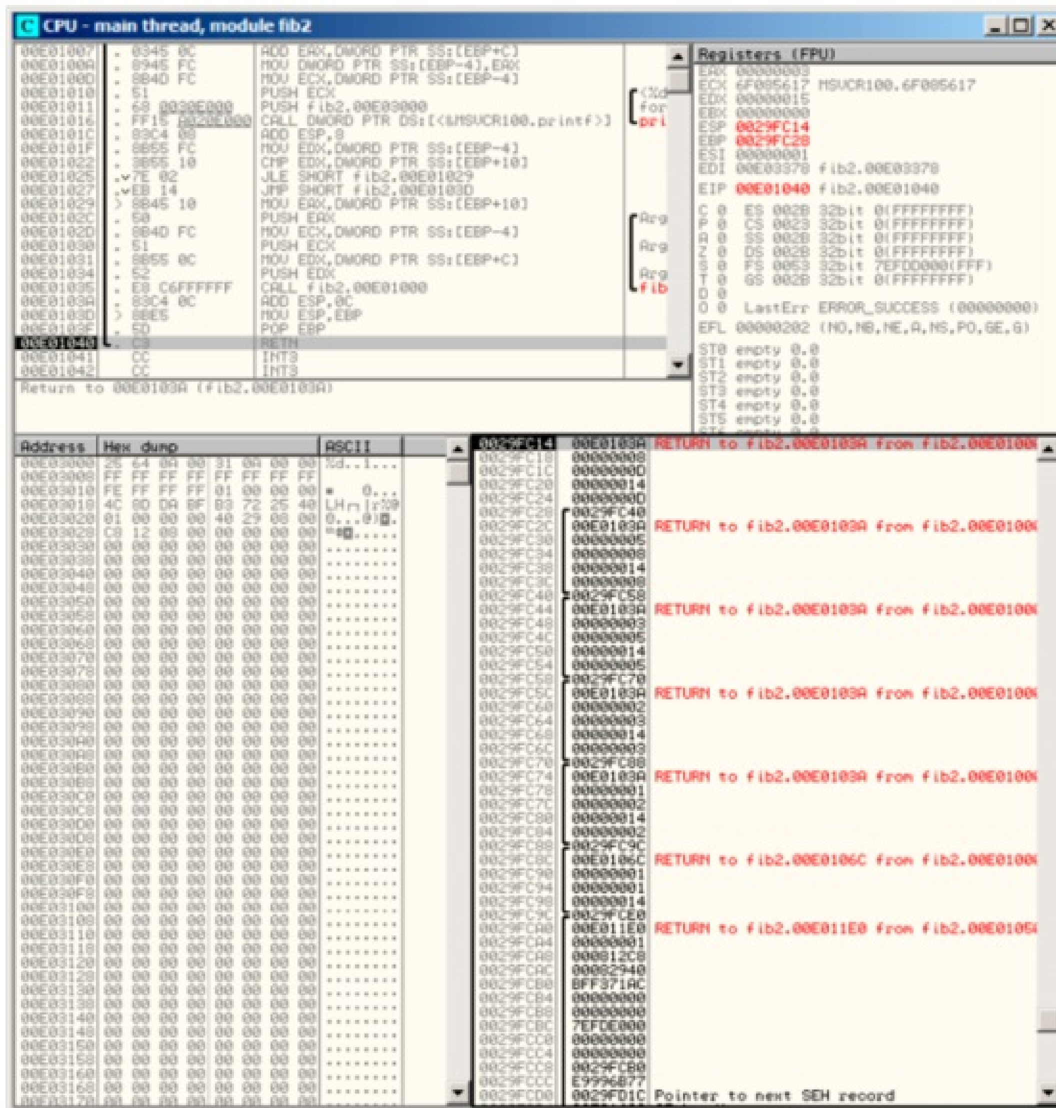


图 36.2: OllyDbg: 最后一次对 `f()` 调用

现在next变量就出现在每一个帧中。

让我们来更加仔细地研究一下栈。作者也向其中加了他的注释：

```
0029FC14  00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC18  00000008 1st argument: a
0029FC1C  0000000D 2nd argument: b
0029FC20  00000014 3rd argument: limit
0029FC24  0000000D "next" variable
0029FC28 /0029FC40 saved EBP register
0029FC2C |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC30 |00000005 1st argument: a
0029FC34 |00000008 2nd argument: b
0029FC38 |00000014 3rd argument: limit
0029FC3C |00000008 "next" variable
0029FC40 ]0029FC58 saved EBP register
0029FC44 |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC48 |00000003 1st argument: a
0029FC4C |00000005 2nd argument: b
0029FC50 |00000014 3rd argument: limit
0029FC54 |00000005 "next" variable
0029FC58 ]0029FC70 saved EBP register
0029FC5C |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC60 |00000002 1st argument: a
0029FC64 |00000003 2nd argument: b
0029FC68 |00000014 3rd argument: limit
0029FC6C |00000003 "next" variable
0029FC70 ]0029FC88 saved EBP register
0029FC74 |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC78 |00000001 1st argument: a           \
0029FC7C |00000002 2nd argument: b           | prepared in f1()
for next f1()
0029FC80 |00000014 3rd argument: limit       /
0029FC84 |00000002 "next" variable
0029FC88 ]0029FC9C saved EBP register
0029FC8C |00E0106C RETURN to fib2.00E0106C from fib2.00E01000
0029FC90 |00000001 1st argument: a           \
0029FC94 |00000001 2nd argument: b           | prepared in main(
) for f1()
0029FC98 |00000014 3rd argument: limit       /
0029FC9C ]0029FCE0 saved EBP register
0029FCA0 |00E011E0 RETURN to fib2.00E011E0 from fib2.00E01050
0029FCA4 |00000001 main() 1st argument: argc \
0029FCA8 |000812C8 main() 2nd argument: argv | prepared in CRT f
or main()
0029FCAC |00082940 main() 3rd argument: envp /
```

在这里我们可以看出：next的值在每次函数调用时都被计算一遍，然后将其作为参数b传递给下一个函数。

36.3 总结

递归函数看起来很nice，但是因为它们对栈的笨重用法在技术上可能会降低性能。所以在写有关性能的关键代码时应该要避免使用递归。

例如，本书的作者曾经写过一个在二叉树中搜寻特定节点的函数。使用递归函数看起来很优雅，但是因为在每次函数调用的开头和结尾会花费额外的时间，它就比使用迭代（不用递归）的情况慢好几倍。

By the way，这是一些函数式PL（Programming language，编程语言，LISP, Python, Lua等）编译器（其中大量使用递归）使用tail call的原因。

第三十七章

CRC32哈希散列计算例子

这是非常流行的CRC32哈希散列计算。

```
/* By Bob Jenkins, (c) 2006, Public Domain */
#include <stdio.h>
#include <stddef.h>
#include <string.h>
typedef unsigned long ub4;
typedef unsigned char ub1;
static const ub4 crctab[256] = {
    0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419,
0x706af48f,
    0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e,
0x97d2d988,
    0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064,
0x6ab020f2,
    0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551,
0x83d385c7,
    0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f,
0x63066cd9,
    0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4,
0xa2677172,
    0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa,
0x42b2986c,
    0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf,
0xabd13d59,
    0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5,
0x56b3c423,
    0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2,
0xb10be924,
    0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190,
0x01db7106,
    0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5,
0xe8b8d433,
    0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb,
0x086d3d2d,
    0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8,
0xf262004e,
    0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6,
0x12b7e950,
    0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3,
0xfbd44c65,
    0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541,
0x3dd895d7,
    0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846,
```

```
0xda60b8d0,
    0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c,
0x270241aa,
    0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409,
0xce61e49f,
    0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17,
0x2eb40d81,
    0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c,
0x74b1d29a,
    0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12,
0x94643b84,
    0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27,
0x7d079eb1,
    0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d,
0x806567cb,
    0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a,
0x67dd4acc,
    0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8,
0xa1d1937e,
    0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd,
0x48b2364b,
    0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3,
0xa867df55,
    0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0,
0x5268e236,
    0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe,
0xb2bd0b28,
    0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b,
0x5bdeae1d,
    0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9,
0xeb0e363f,
    0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae,
0x0cb61b38,
    0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4,
0xf1d4e242,
    0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1,
0x18b74777,
    0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff,
0xf862ae69,
    0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354,
0x3903b3c2,
    0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a,
0xd9d65adc,
    0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f,
0x30b5ffe9,
    0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605,
0xcdd70693,
    0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02,
0x2a6f2b94,
    0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
};
/* how to derive the values in crctab[] from polynomial 0xedb883
20 */
```

```
void build_table()
{
    ub4 i, j;
    for (i=0; i<256; ++i) {
        j = i;
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        printf("0x%.8lx, ", j);
        if (i%6 == 5) printf("");
     }
}
/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
{
    ub4 i;
    const ub1 *k = key;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
    return hash;
}

/* To use, try "gcc -O crc.c -o crc; crc < crc.c" */
int main()
{
    char s[1000];
    while (gets(s)) printf("%.8lx", crc(s, strlen(s), 0));
    return 0;
}
```

我们只关心crc()函数。注意for()语句两个循环初始化：hash=len,i=0。标准C/C++允许这样做。循环体内通常需要使用两个初始化部分。 让我们用MSVC优化（/Ox）。为了简洁，仅列出crc()函数的代码，包括我做的注释。

```
key$ = 8                ; size = 4
_len$ = 12               ; size = 4
_hash$ = 16              ; size = 4
_crc PROC
    mov     edx, DWORD PTR _len$[esp-4]
    xor     ecx, ecx    ; i will be stored in ECX
    mov     eax, edx
    test    edx, edx
    jbe     SHORT $LN1@crc
    push    ebx
    push    esi
    mov     esi, DWORD PTR _key$[esp+4] ; ESI = key
    push    edi
$LL3@crc:

; work with bytes using only 32-bit registers. byte from address
 key+i we store into EDI

    movzx   edi, BYTE PTR [ecx+esi]
    mov     ebx, eax ; EBX = (hash = len)
    and     ebx, 255 ; EBX = hash & 0xff
; XOR EDI, EBX (EDI=EDI^EBX) - this operation uses all 32 bits o
f each register
; but other bits (8-31) are cleared all time, so it’s OK
; these are cleared because, as for EDI, it was done by MOVZX in
struction above
; high bits of EBX was cleared by AND EBX, 255 instruction above
 (255 = 0xff)

    xor     edi, ebx

; EAX=EAX>>8; bits 24-31 taken "from nowhere" will be cleared

        shr     eax, 8

; EAX=EAX^crctab[EDI*4] - choose EDI-th element from crctab[] ta
ble
    xor     eax, DWORD PTR _crctab[edi*4]
    inc     ecx          ; i++
    cmp     ecx, edx ; i<len ?
    jb      SHORT $LL3@crc ; yes
    pop     edi
    pop     esi
    pop     ebx
$LN1@crc:
    ret 0
_crc ENDP
```

我们来看GCC 4.4.1优化后的代码：

```
    public crc
crc     proc near
key     = dword ptr 8
hash    = dword ptr 0Ch
push    ebp
xor     edx, edx
mov     ebp, esp
push    esi
mov     esi, [ebp+key]
push    ebx
mov     ebx, [ebp+hash]
test    ebx, ebx
mov     eax, ebx
jz      short loc_80484D3
nop                 ; padding
lea     esi, [esi+0]        ; padding; ESI doesn’t changing here
loc_80484B8:
mov     ecx, eax        ; save previous state of hash to ECX
xor     al, [esi+edx]       ; AL=*(key+i)
add     edx, 1          ; i++
shr     ecx, 8          ; ECX=hash>>8
movzx   eax, al         ; EAX=*(key+i)
mov     eax, dword ptr ds:crctab[eax*4]     ; EAX=crctab[EAX]
xor     eax, ecx        ; hash=EAX^ECX
cmp     ebx, edx
ja      short loc_80484B8
loc_80484D3:
    pop ebx
    pop esi
    pop ebp
    retn
crc endp
```

GCC在循环开始的时候通过填入NOP和lea esi,esi+0来按8字节对齐。更多信息请阅读npad小结（64）。

#

网址的计算实例

#

循环:几个迭代器

#

Duff’s device

#

除以9

#

将字符串转化为数字(atoi())

第四十三章

内联函数

内联代码是指当编译的时候，将函数体直接嵌入正确位置，而不是在这个位置放上函数声明。

```
#include <stdio.h>
int celsius_to_fahrenheit (int celsius)
{
    return celsius * 9 / 5 + 32;
};
int main(int argc, char *argv[])
{
    int celsius=atol(argv[1]);
    printf ("%d\n", celsius_to_fahrenheit (celsius));
};
```

这个编译是意料之中的，但是如果换成GCC的优化方案，我们会看到：

清单43.2: GCC 4.8.1 -O3

```
_main:
    push ebp
    mov ebp, esp
    and esp, -16
    sub esp, 16
    call ___main
    mov eax, DWORD PTR [ebp+12]
    mov eax, DWORD PTR [eax+4]
    mov DWORD PTR [esp], eax
    call _atol
    mov edx, 1717986919
    mov DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d\12\0"
    lea ecx, [eax+eax*8]
    mov eax, ecx
    imul edx
    sar ecx, 31
    sar edx
    sub edx, ecx
    add edx, 32
    mov DWORD PTR [esp+4], edx
    call _printf
    leave
    ret
```

这里的除法由乘法完成。 是的，我们的小函数被放到了printf()调用之前。为什么？因为这比直接执行函数之前的“调用/返回”过程速度更快。 在过去，这样的函数在函数声明的时候必须被标记为“内联”。在现代，这样的函数会自动被编译器识别。 另外一个普通的自动优化的例子是内联字符串函数，比如strcpy(),strcmp()等

43.1 字符串和内存函数

43.1.1 strcmp()

清单27.3 : 另一个简单的例子

```
bool is_bool (char *s)
{
    if (strcmp (s, "true")==0)
    return true;
    if (strcmp (s, "false")==0)
    return false;
    assert(0);
};
```

清单27.4： GCC 4.8.1 -O3

```
_is_bool:
    push edi
    mov ecx, 5
    push esi
    mov edi, OFFSET FLAT:LC0 ; "true\0"
    sub esp, 20
    mov esi, DWORD PTR [esp+32]
    repz cmpsb
    je L3
    mov esi, DWORD PTR [esp+32]
    mov ecx, 6
    mov edi, OFFSET FLAT:LC1 ; "false\0"
    repz cmpsb
    seta cl
    setb dl
    xor eax, eax
    cmp cl, dl
    jne L8
    add esp, 20
    pop esi
    pop edi
    ret
```

这是一个经常可以见到的关于MSVC生成的strcmp()的例子。

清单27.5: MSVC

```
    mov dl, [eax]
    cmp dl, [ecx]
    jnz short loc_10027FA0
    test dl, dl
    jz short loc_10027F9C
    mov dl, [eax+1]
    cmp dl, [ecx+1]
    jnz short loc_10027FA0
    add eax, 2
    add ecx, 2
    test dl, dl
    jnz short loc_10027F80
    loc_10027F9C: ; CODE XREF: f1+448
    xor eax, eax
    jmp short loc_10027FA5
; ---------------------------------------------------------------
-------------
    loc_10027FA0: ; CODE XREF: f1+444
; f1+450
    sbb eax, eax
    sbb eax, 0FFFFFFFFh
```

43.1.2 strlen()

43.1.3 strcpy()

43.1.4 memset()

Example#1

Example#2

43.1.5 memcpy()

Short blocks

Long blocks

43.1.6 memcmp()

43.1.7 IDA script

我写了一个小的用于搜索和归纳的IDA脚本，这样的脚本经常能在内联代码中看到：IDA_scripts.

C99的限制

这个例子说明了为什么某些情况下FORTRAN的速度比C/C++要快

```
void f1 (int* x, int* y, int* sum, int* product, int* sum_produc
t, int* update_me, size_t s)
{
    for (int i=0; i<s; i++)
        {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    };
};
```

这是一个十分简单的例子，但是有一点需要注意：指向update_me数组的指针也可以指向sum数组，甚至是sum_product数组。但是这不是严重的错误，对吗？ 编译器很清楚这一点，所以他在循环体中产生了四个阶段： 1.计算下一个sum[i] 2.计算下一个product[i] 3.计算下一个unpdate_me[i] 4.计算下一个sum_product[i],在这个阶段，我们需要从已经计算过sum[i]和product[i]的内存中载入数据

最后一个阶段可以优化吗？既然已经计算过的sum[i]和product[i]是不需要再次从内存装载的（因为我们已经计算过他们了）。但是编译器不能保证在第三个阶段没有东西被覆盖掉！这就叫“指针别名”，在这种情况下编译器无法确定指针指向区域的内存是否已经被改变。

C99标准中的限制给解决这一问题带来了一线曙光。由设计器传送给编译器的函数单元在标记这种关键字(restrict)后，它会指向不同的内存区域，并且不 会被混用。如果要更加准确地描述这种情况，restrict表明了只有指针是可以访问对象的。这样的话我们可以通过特定的指针进行工作，并且不会用到其他指针。也就是说一个对象如果被标记为restrict，那么它只能通过一个指针访问。 我们把每个指向变量的指针标记为restrict关键字：

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, in
t* restrict product, int*
restrict sum_product,
int* restrict update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    };
};
```

来看下结果：

清单44.1： GCC x64: f1()

```
f1:
    push r15 r14 r13 r12 rbp rdi rsi rbx
    mov r13, QWORD PTR 120[rsp]
    mov rbp, QWORD PTR 104[rsp]
    mov r12, QWORD PTR 112[rsp]
    test r13, r13
    je .L1
    add r13, 1
    xor ebx, ebx
    mov edi, 1
    xor r11d, r11d
    jmp .L4
    .L6:
    mov r11, rdi
    mov rdi, rax
    .L4:
    lea rax, 0[0+r11*4]
    lea r10, [rcx+rax]
    lea r14, [rdx+rax]
    lea rsi, [r8+rax]
    add rax, r9
    mov r15d, DWORD PTR [r10]
    add r15d, DWORD PTR [r14]
    mov DWORD PTR [rsi], r15d ; store to sum[]
    mov r10d, DWORD PTR [r10]
    imul r10d, DWORD PTR [r14]
    mov DWORD PTR [rax], r10d ; store to product[]
    mov DWORD PTR [r12+r11*4], ebx ; store to update_me[]
    add ebx, 123
    mov r10d, DWORD PTR [rsi] ; reload sum[i]
    add r10d, DWORD PTR [rax] ; reload product[i]
    lea rax, 1[rdi]
    cmp rax, r13
    mov DWORD PTR 0[rbp+r11*4], r10d ; store to sum_product[]
    jne .L6
    .L1:
    pop rbx rsi rdi rbp r12 r13 r14 r15
    ret
```

清单44.2： GCC x64: f2()

```
f2:
    push r13 r12 rbp rdi rsi rbx
    mov r13, QWORD PTR 104[rsp]
    mov rbp, QWORD PTR 88[rsp]
    mov r12, QWORD PTR 96[rsp]
    test r13, r13
    je .L7
    add r13, 1
    xor r10d, r10d
    mov edi, 1
    xor eax, eax
    jmp .L10
    .L11:
    mov rax, rdi
    mov rdi, r11
    .L10:
    mov esi, DWORD PTR [rcx+rax*4]
    mov r11d, DWORD PTR [rdx+rax*4]
    mov DWORD PTR [r12+rax*4], r10d ; store to update_me[]
    add r10d, 123
    lea ebx, [rsi+r11]
    imul r11d, esi
    mov DWORD PTR [r8+rax*4], ebx ; store to sum[]
    mov DWORD PTR [r9+rax*4], r11d ; store to product[]
    add r11d, ebx
    mov DWORD PTR 0[rbp+rax*4], r11d ; store to sum_product[]
    lea r11, 1[rdi]
    cmp r11, r13
    jne .L11
    .L7:
    pop rbx rsi rdi rbp r12 r13
    ret
```

被编译过的f1()和f2()的不同点是：在f1()中，sum[i]和product[i]在循环中途被装入，但是在f2()中没有这样的特性。已经计算过的变量将被使用，既然我们已经向编译器"保证"在循环执行期间，sum[i]和product[i]不会发生改变，所以编译器"确信"变量的值不用从内存被再装入。很明显，第二个例子的程序更快。 但是如果函数变量中的指针发生混淆的情况又能如何呢？这与一个程序员的认知有关，并且结果是不正确的。 回到FORTRAN。FORTRAN语言编译器按照指针的本身含义对待他，所以当FORTRAN程序在这种情况下不可能使用restrict的时候，它可以生成生成执行更快的代码。

这有什么实用价值？当函数处理内存中很多大"块"的时候，比如说用超级计算机解决线性代数问题。或许这就是为什么FORTRAN语言还在这个领域被使用。 但是当迭代步骤不是很多的时候，速度的增加并不是显著的。

第四十九章

处理不当的反汇编代码

逆向工程师经常需要处理不当的反汇编代码

49.1反汇编于不正确起始位置(x86)

不同于ARM和MIPS架构(任何指令长度只有2个字节长度或者4个字节长度)， x86架构的指令长度是不定长的，因此，任何反汇编器从x86指令中间开始反汇编，可能会长生不正确的结果。

举个例子：

```
add [ebp-31F7Bh], cl
dec dword ptr [ecx-3277Bh]
dec dword ptr [ebp-2CF7Bh]
inc dword ptr [ebx-7A76F33Ch]
fdiv st(4), st
;-------------------------------------------------------------
db 0FFh
;-------------------------------------------------------------
dec dword ptr [ecx-21F7Bh]
dec dword ptr [ecx-22373h]
dec dword ptr [ecx-2276Bh]
dec dword ptr [ecx-22B63h]
dec dword ptr [ecx-22F4Bh]
dec dword ptr [ecx-23343h]
jmp dword ptr [esi-74h]
;-------------------------------------------------------------
xchg eax, ebp
clc
std
;-------------------------------------------------------------
db 0FFh
db 0FFh
;-------------------------------------------------------------
mov word ptr [ebp-214h], cs
mov word ptr [ebp-238h], ds
mov word ptr [ebp-23Ch], es
mov word ptr [ebp-240h], fs
mov word ptr [ebp-244h], gs
pushf
pop dword ptr [ebp-210h]
mov eax, [ebp+4]
mov [ebp-218h], eax
lea eax, [ebp+4]
mov [ebp-20Ch], eax
mov dword ptr [ebp-2D0h], 10001h
mov eax, [eax-4]
mov [ebp-21Ch], eax
mov eax, [ebp+0Ch]
mov [ebp-320h], eax
mov eax, [ebp+10h]
mov [ebp-31Ch], eax
mov eax, [ebp+4]
mov [ebp-314h], eax
call ds:IsDebuggerPresent
mov edi, eax
lea eax, [ebp-328h]
push eax
call sub_407663
pop ecx
test eax, eax
jnz short loc_402D7B
```

虽然上面的代码片段一开始是从错误的起始位置反汇编的，但最终，反汇编器能够自己调整到正确的轨道上。

49.2 不正确的反汇编代码的特点

可以很容易发现它们的共同特点是：

很少出现大尺寸的指令，最常见的有x86指令的push，mov，call。 但是我们可以看到这些指令来自各个不同的指令组，有FPU指令，IN/OUT指令，少数的系统指令，一切都是因为反汇编器从一个错误的位置上开始反汇编机器码给搞砸了。 偏移量和立即数都是一些随机值，而且数值较大。 跳转到不正确的偏移地址常常会跳转到另一个指令的中间。 代码清单28.1:x86架构不正确的反汇编代码示例

```
mov     bl, 0Ch
mov     ecx, 0D38558Dh
mov     eax, ds:2C869A86h
db      67h
mov     dl, 0CCh
insb
movsb
push    eax
xor     [edx-53h], ah
fcom    qword ptr [edi-45A0EF72h]
pop     esp
pop     ss
in      eax, dx
dec     ebx
push    esp
lds     esp, [esi-41h]
retf
rcl     dword ptr [eax], cl
mov     cl, 9Ch
mov     ch, 0DFh
push    cs
insb
mov     esi, 0D9C65E4Dh
imul    ebp, [ecx], 66h
pushf
sal     dword ptr [ebp-64h], cl
sub     eax, 0AC433D64h
out     8Ch, eax
pop     ss
sbb     [eax], ebx
aas
xchg    cl, [ebx+ebx*4+14B31Eh]
jecxz   short near ptr loc_58+1
xor     al, 0C6h
inc     edx
db      36h
pusha
```

```
    stosb
    test    [ebx], ebx
    sub     al, 0D3h ; 'L'
    pop     eax
    stosb

loc_58: ; CODE XREF: seg000:0000004A
    test    [esi], eax
    inc     ebp
    das
    db      64h
    pop     ecx
    das
    hlt

    pop     edx
    out     0B0h, al
    lodsb
    push    ebx
    cdq
    out     dx, al
    sub     al, 0Ah
    sti
    outsd
    add     dword ptr [edx], 96FCBE4Bh
    and     eax, 0E537EE4Fh
    inc     esp
    stosd
    cdq
    push    ecx
    in      al, 0CBh
    mov     ds:0D114C45Ch, al
    mov     esi, 659D1985h
    enter   6FE8h, 0D9h
    enter   6FE6h, 0D9h
    xchg    eax, esi
    sub     eax, 0A599866Eh
    retn

    pop     eax
    dec     eax
    adc     al, 21h ; '!'
    lahf
    inc     edi
    sub     eax, 9062EE5Bh
    bound   eax, [ebx]

loc_A2: ; CODE XREF: seg000:00000120
    wait
    iret

    jnb     short loc_D7
    cmpsd
```

```
    iret

    jnb     short loc_D7
    sub     ebx, [ecx]
    in      al, 0Ch
    add     esp, esp
    mov     bl, 8Fh
    xchg    eax, ecx
    int     67h
    pop     ds
    pop     ebx
    db      36h
    xor     esi, [ebp-4Ah]
    mov     ebx, 0EB4F980Ch
    repne add bl, dh
    imul    ebx, [ebp+5616E7A5h], 67A4D1EEh
    xchg    eax, ebp
    scasb
    push    esp
    wait
    mov     dl, 11h
    mov     ah, 29h ; ')'
    fist    dword ptr [edx]

loc_D7: ; CODE XREF: seg000:000000A4
        ; seg000:000000A8 ...
    dec     dword ptr [ebp-5D0E0BA4h]
    call    near ptr 622FEE3Eh
    sbb     ax, 5A2Fh
    jmp     dword ptr cs:[ebx]

    xor     ch, [edx-5]
    inc     esp
    push    edi
    xor     esp, [ebx-6779D3B8h]
    pop     eax
    int     3               ; Trap to Debugger
    rcl     byte ptr [ebx-3Eh], cl
    xor     [edi], bl
    sbb     al, [edx+ecx*4]
    xor     ah, [ecx-1DA4E05Dh]
    push    edi
    xor     ah, cl
    popa
    cmp     dword ptr [edx-62h], 46h ; 'F'
    dec     eax
    in      al, 69h
    dec     ebx
    iret

    or      al, 6
    jns     short near ptr loc_D7+3
    shl     byte ptr [esi], 42h
```

```
    repne adc [ebx+2Ch], eax
    icebp
    cmpsd
    leave
    push    esi
    jmp     short loc_A2

    and     eax, 0F2E41FE9h
    push    esi
    loop    loc_14F
    add     ah, fs:[edx]

loc_12D: ; CODE XREF: seg000:00000169
    mov     dh, 0F7h
    add     [ebx+7B61D47Eh], esp
    mov     edi, 79F19525h
    rcl     byte ptr [eax+22015F55h], cl
    cli
    sub     al, 0D2h ; 'T'
    dec     eax
    mov     ds:0A81406F5h, eax
    sbb     eax, 0A7AA179Ah
    in      eax, dx

loc_14F: ; CODE XREF: seg000:00000128
    and     [ebx-4CDFAC74h], ah
    pop     ecx
    push    esi
    mov     bl, 2Dh ; '-'
    in      eax, 2Ch
    stosd
    inc     edi
    push    esp

locret_15E: ; CODE XREF: seg000:loc_1A0
    retn    0C432h

    and     al, 86h
    cwde
    and     al, 8Fh
    cmp     ebp, [ebp+7]
    jz      short loc_12D
    sub     bh, ch
    or      dword ptr [edi-7Bh], 8A16C0F7h
    db      65h
    insd
    mov     al, ds:0A3A5173Dh
    dec     ecx
    push    ds
    xor     al, cl
    jg      short loc_195
    push    6Eh ; 'n'
    out     0DDh, al
```

```
    inc     edi
    sub     eax, 6899BBF1h
    leave
    rcr     dword ptr [ecx-69h], cl
    sbb     ch, [edi+5EDDCB54h]

loc_195: ; CODE XREF: seg000:0000017F
    push    es
    repne sub ah, [eax-105FF22Dh]
    cmc
    and     ch, al

loc_1A0: ; CODE XREF: seg000:00000217
    jnp     short near ptr locret_15E+1
    or      ch, [eax-66h]
    add     [edi+edx-35h], esi
    out     dx, al
    db      2Eh
    call    far ptr 1AAh:6832F5DDh
    jz      short near ptr loc_1DA+1
    sbb     esp, [edi+2CB02CEFh]
    xchg    eax, edi
    xor     [ebx-766342ABh], edx

loc_1C1: ; CODE XREF: seg000:00000212
    cmp     eax, 1BE9080h
    add     [ecx], edi
    aad     0
    imul    esp, [edx-70h], 0A8990126h
    or      dword ptr [edx+10C33693h], 4Bh
    popf

loc_1DA: ; CODE XREF: seg000:000001B2
    mov     ecx, cs
    aaa
    mov     al, 39h ; '9'
    adc     byte ptr [eax-77F7F1C5h], 0C7h
    add     [ecx], bl
    retn    0DD42h

    db      3Eh
    mov     fs:[edi], edi
    and     [ebx-24h], esp
    db      64h
    xchg    eax, ebp
    push    cs
    adc     eax, [edi+36h]
    mov     bh, 0C7h
    sub     eax, 0A710CBE7h
    xchg    eax, ecx
    or      eax, 51836E42h
    xchg    eax, ebx
    inc     ecx
```

```
    jb      short near ptr loc_21E+3
    db      64h
    xchg    eax, esp
    and     dh, [eax-31h]
    mov     ch, 13h
    add     ebx, edx
    jnb     short loc_1C1
    db      65h
    adc     al, 0C5h
    js      short loc_1A0
    sbb     eax, 887F5BEEh

loc_21E: ; CODE XREF: seg000:00000207
    mov     eax, 888E1FD6h
    mov     bl, 90h
    cmp     [eax], ecx
    rep int 61h             ; reserved for user interrupt
    and     edx, [esi-7EB5C9EAh]
    fisttp  qword ptr [eax+esi*4+38F9BA6h]
    jmp     short loc_27C

    fadd    st, st(2)
    db      3Eh
    mov     edx, 54C03172h
    retn

    db      64h
    pop     ds
    xchg    eax, esi
    rcr     ebx, cl
    cmp     [di+2Eh], ebx
    repne xor [di-19h], dh
    insd
    adc     dl, [eax-0C4579F7h]
    push    ss
    xor     [ecx+edx*4+65h], ecx
    mov     cl, [ecx+ebx-32E8AC51h]
    or      [ebx], ebp
    cmpsb
    lodsb
    iret
```

代码清单28.2:x86_64架构不正确的反汇编代码示例

```
    lea     esi, [rax+rdx*4+43558D29h]

loc_AF3: ; CODE XREF: seg000:0000000000000B46
    rcl     byte ptr [rsi+rax*8+29BB423Ah], 1
    lea     ecx, cs:0FFFFFFFFB2A6780Fh
    mov     al, 96h
    mov     ah, 0CEh
```

```
    push    rsp
    lods    byte ptr [esi]

    db  2Fh ; /

    pop     rsp
    db      64h
    retf    0E993h

    cmp     ah, [rax+4Ah]
    movzx   rsi, dword ptr [rbp-25h]
    push    4Ah
    movzx   rdi, dword ptr [rdi+rdx*8]

    db  9Ah

    rcr     byte ptr [rax+1Dh], cl
    lodsd
    xor     [rbp+6CF20173h], edx
    xor     [rbp+66F8B593h], edx
    push    rbx
    sbb     ch, [rbx-0Fh]
    stosd
    int     87h
    db      46h, 4Ch
    out     33h, rax
    xchg    eax, ebp
    test    ecx, ebp
    movsd
    leave
    push    rsp

    db  16h

    xchg    eax, esi
    pop     rdi

loc_B3D: ; CODE XREF: seg000:0000000000000B5F
    mov     ds:93CA685DF98A90F9h, eax
    jnz     short near ptr loc_AF3+6
    out     dx, eax
    cwde
    mov     bh, 5Dh ; ']'
    movsb
    pop     rbp

    db  60h ; `

    movsxd  rbp, dword ptr [rbp-17h]
    pop     rbx
    out     7Dh, al
    add     eax, 0D79BE769h
```

```
    db  1Fh

    retf    0CAB9h

    jl      short near ptr loc_B3D+4
    sal     dword ptr [rbx+rbp+4Dh], 0D3h
    mov     cl, 41h ; 'A'
    imul    eax, [rbp-5B77E717h], 1DDE6E5h
    imul    ecx, ebx, 66359BCCh
    xlat

    db  60h ; `

    cmp     bl, [rax]
    and     ebp, [rcx-57h]
    stc
    sub     [rcx+1A533AB4h], al
    jmp     short loc_C05

    db  4Bh ; K

    int     3               ; Trap to Debugger
    xchg    ebx, [rsp+rdx-5Bh]

    db 0D6h

    mov     esp, 0C5BA61F7h
    out     0A3h, al        ; Interrupt Controller #2, 8259A
    add     al, 0A6h
    pop     rbx
    cmp     bh, fs:[rsi]
    and     ch, cl
    cmp     al, 0F3h

    db  0Eh

    xchg    dh, [rbp+rax*4-4CE9621Ah]
    stosd
    xor     [rdi], ebx
    stosb
    xchg    eax, ecx
    push    rsi
    insd
    fidiv   word ptr [rcx]
    xchg    eax, ecx
    mov     dh, 0C0h ; 'L'
    xchg    eax, esp
    push    rsi
    mov     dh, [rdx+rbp+6918F1F3h]
    xchg    eax, ebp
    out     9Dh, al

loc_BC0: ; CODE XREF: seg000:0000000000000C26
```

```
    or      [rcx-0Dh], ch
    int     67h             ;  - LIM EMS
    push    rdx
    sub     al, 43h ; 'C'
    test    ecx, ebp
    test    [rdi+71F372A4h], cl

    db    7

    imul    ebx, [rsi-0Dh], 2BB30231h
    xor     ebx, [rbp-718B6E64h]
    jns     short near ptr loc_C56+1
    ficomp  dword ptr [rcx-1Ah]
    and     eax, 69BEECC7h
    mov     esi, 37DA40F6h
    imul    r13, [rbp+rdi*8+529F33CDh], 0FFFFFFFFF35CDD30h
    or      [rbx], edx
    imul    esi, [rbx-34h], 0CDA42B87h

    db  36h ; 6
    db  1Fh


loc_C05: ; CODE XREF: seg000:0000000000000B86
    add     dh, [rcx]
    mov     edi, 0DD3E659h
    ror     byte ptr [rdx-33h], cl
    xlat
    db      48h
    sub     rsi, [rcx]

    db  1Fh
    db    6

    xor     [rdi+13F5F362h], bh
    cmpsb
    sub     esi, [rdx]
    pop     rbp
    sbb     al, 62h ; 'b'
    mov     dl, 33h ; '3'

    db  4Dh ; M
    db  17h

    jns     short loc_BC0
    push    0FFFFFFFFFFFFFF86h

loc_C2A: ; CODE XREF: seg000:0000000000000C8F
    sub     [rdi-2Ah], eax

    db 0FEh

    cmpsb
```

```
    wait
    rcr     byte ptr [rax+5Fh], cl
    cmp     bl, al
    pushfq
    xchg    ch, cl

    db  4Eh ; N
    db  37h ; 7

    mov     ds:0E43F3CCD3D9AB295h, eax
    cmp     ebp, ecx
    jl      short loc_C87
    retn    8574h

    out     3, al           ; DMA controller, 8237A-5.
                            ; channel 1 base address and word co
unt

loc_C4C: ; CODE XREF: seg000:0000000000000C7F
    cmp     al, 0A6h
    wait
    push    0FFFFFFFFFFFFFFBEh

    db  82h

    ficom   dword ptr [rbx+r10*8]

loc_C56: ; CODE XREF: seg000:0000000000000BDE
    jnz     short loc_C76
    xchg    eax, edx
    db      26h
    wait
    iret

    push    rcx

    db  48h ; H
    db  9Bh
    db  64h ; d
    db  3Eh ; >
    db  2Fh ; /

    mov     al, ds:8A7490CA2E9AA728h
    stc

    db  60h ; `

    test    [rbx+rcx], ebp
    int     3               ; Trap to Debugger
    xlat

loc_C72: ; CODE XREF: seg000:0000000000000CC6
    mov     bh, 98h
```

```
    db  2Eh ; .
    db 0DFh


loc_C76: ; CODE XREF: seg000:loc_C56
    jl      short loc_C91
    sub     ecx, 13A7CCF2h
    movsb
    jns     short near ptr loc_C4C+1
    cmpsd
    sub     ah, ah
    cdq

    db  6Bh ; k
    db  5Ah ; Z


loc_C87: ; CODE XREF: seg000:0000000000000C45
    or      ecx, [rbx+6Eh]
    rep in eax, 0Eh         ; DMA controller, 8237A-5.
                            ; Clear mask registers.
                            ; Any OUT enables all 4 channels.
    cmpsb
    jnb     short loc_C2A

loc_C91: ; CODE XREF: seg000:loc_C76
    scasd
    add     dl, [rcx+5FEF30E6h]
    enter   0FFFFFFFFFFFFC733h, 7Ch
    insd
    mov     ecx, gs
    in      al, dx
    out     2Dh, al
    mov     ds:6599E434E6D96814h, al
    cmpsb
    push    0FFFFFFFFFFFFFFD6h
    popfq
    xor     ecx, ebp
    db      48h
    insb
    test    al, cl
    xor     [rbp-7Bh], cl
    and     al, 9Bh

    db  9Ah

    push    rsp
    xor     al, 8Fh
    cmp     eax, 924E81B9h
    clc
    mov     bh, 0DEh
    jbe     short near ptr loc_C72+1
```

```
    db  1Eh

    retn    8FCAh

    db 0C4h ; -


loc_CCD: ; CODE XREF: seg000:0000000000000D22
    adc     eax, 7CABFBF8h

    db  38h ; 8

    mov     ebp, 9C3E66FCh
    push    rbp
    dec     byte ptr [rcx]
    sahf
    fidivr  word ptr [rdi+2Ch]

    db  1Fh

    db      3Eh
    xchg    eax, esi

loc_CE2: ; CODE XREF: seg000:0000000000000D5E
    mov     ebx, 0C7AFE30Bh
    clc
    in      eax, dx
    sbb     bh, bl
    xchg    eax, ebp

    db  3Fh ; ?

    cmp     edx, 3EC3E4D7h
    push    51h
    db      3Eh
    pushfq
    jl      short loc_D17
    test    [rax-4CFF0D49h], ebx

    db  2Fh ; /

    rdtsc
    jns     short near ptr loc_D40+4
    mov     ebp, 0B2BB03D8h
    in      eax, dx

    db  1Eh

    fsubr   dword ptr [rbx-0Bh]
    jns     short loc_D70
    scasd
    mov     ch, 0C1h ; '+'
```

```
    add     edi, [rbx-53h]

    db 0E7h


loc_D17: ; CODE XREF: seg000:0000000000000CF7
    jp      short near ptr unk_D79
    scasd
    cmc
    sbb     ebx, [rsi]
    fsubr   dword ptr [rbx+3Dh]
    retn

    db    3

    jnp     short near ptr loc_CCD+4
    db      36h
    adc     r14b, r13b

    db  1Fh

    retf

    test    [rdi+rdi*2], ebx
    cdq
    or      ebx, edi
    test    eax, 310B94BCh
    ffreep  st(7)
    cwde
    sbb     esi, [rdx+53h]
    push    5372CBAAh

loc_D40: ; CODE XREF: seg000:0000000000000D02
    push    53728BAAh
    push    0FFFFFFFFF85CF2FCh

    db  0Eh

    retn    9B9Bh

    movzx   r9, dword ptr [rdx]
    adc     [rcx+43h], ebp
    in      al, 31h

    db  37h ; 7

    jl      short loc_DC5
    icebp
    sub     esi, [rdi]
    clc
    pop     rdi
    jb      short near ptr loc_CE2+1
    or      al, 8Fh
```

```
    mov     ecx, 770EFF81h
    sub     al, ch
    sub     al, 73h ; 's'
    cmpsd
    adc     bl, al
    out     87h, eax        ; DMA page register 74LS612:
                            ; Channel 0 (address bits 16-23)

loc_D70: ; CODE XREF: seg000:0000000000000D0E
    adc     edi, ebx
    db      49h
    outsb
    enter   33E5h, 97h
    xchg    eax, ebx

unk_D79   db 0FEh ; CODE XREF: seg000:loc_D17
          db 0BEh
          db 0E1h
          db  82h


loc_D7D: ; CODE XREF: seg000:0000000000000DB3
    cwde

    db    7
    db  5Ch ; \
    db  10h
    db  73h ; s
    db 0A9h
    db  2Bh ; +
    db  9Fh


loc_D85: ; CODE XREF: seg000:0000000000000DD1
    dec     dh
    jnz     short near ptr loc_DD3+3
    mov     ds:7C1758CB282EF9BFh, al
    sal     ch, 91h
    rol     dword ptr [rbx+7Fh], cl
    fbstp   tbyte ptr [rcx+2]
    repne mov al, ds:4BFAB3C3ECF2BE13h
    pushfq
    imul    edx, [rbx+rsi*8+3B484EE9h], 8EDC09C6h
    cmp     [rax], al
    jg      short loc_D7D
    xor     [rcx-638C1102h], edx
    test    eax, 14E3AD7h
    insd

    db  38h ; 8
    db  80h
    db 0C3h
```

```
loc_DC5: ; CODE XREF: seg000:0000000000000D57
         ; seg000:0000000000000DD8
    cmp     ah, [rsi+rdi*2+527C01D3h]
    sbb     eax, 5FC631F0h
    jnb     short loc_D85

loc_DD3: ; CODE XREF: seg000:0000000000000D87
    call    near ptr 0FFFFFFFFC03919C7h
    loope   near ptr loc_DC5+3
    sbb     al, 0C8h
    std
```

代码清单28.2:ARM架构(ARM 模式)不正确的反汇编代码示例

```
BLNE    0xFE16A9D8
BGE     0x1634D0C
SVCCS   0x450685
STRNVT  R5, [PC],#-0x964
LDCGE   p6, c14, [R0],#0x168
STCCSL  p9, c9, [LR],#0x14C
CMNHIP  PC, R10,LSL#22
FLDMIADNV LR!, {D4}
MCR     p5, 2, R2,c15,c6, 4
BLGE    0x1139558
BLGT    0xFF9146E4
STRNEB  R5, [R4],#0xCA2
STMNEIB R5, {R0,R4,R6,R7,R9-SP,PC}
STMIA   R8, {R0,R2-R4,R7,R8,R10,SP,LR}^
STRB    SP, [R8],PC,ROR#18
LDCCS   p9, c13, [R6,#0x1BC]
LDRGE   R8, [R9,#0x66E]
STRNEB  R5, [R8],#-0x8C3
STCCSL  p15, c9, [R7,#-0x84]
RSBLS   LR, R2, R11,ASR LR
SVCGT   0x9B0362
SVCGT   0xA73173
STMNEDB R11!, {R0,R1,R4-R6,R8,R10,R11,SP}
STR     R0, [R3],#-0xCE4
LDCGT   p15, c8, [R1,#0x2CC]
LDRCCB  R1, [R11],-R7,ROR#30
BLLT    0xFED9D58C
BL      0x13E60F4
LDMVSIB R3!, {R1,R4-R7}^
USATNE  R10, #7, SP,LSL#11
LDRGEB  LR, [R1],#0xE56
STRPLT  R9, [LR],#0x567
LDRLT   R11, [R1],#-0x29B
SVCNV   0x12DB29
MVNNVS  R5, SP,LSL#25
LDCL    p8, c14, [R12,#-0x288]
STCNEL  p2, c6, [R6,#-0xBC]!
SVCNV   0x2E5A2F
BLX     0x1A8C97E
TEQGE   R3, #0x1100000
STMLSIA R6, {R3,R6,R10,R11,SP}
BICPLS  R12, R2, #0x5800
BNE     0x7CC408
TEQGE   R2, R4,LSL#20
SUBS    R1, R11, #0x28C
BICVS   R3, R12, R7,ASR R0
LDRMI   R7, [LR],R3,LSL#21
BLMI    0x1A79234
STMVCDB R6, {R0-R3,R6,R7,R10,R11}
EORMI   R12, R6, #0xC5
MCRRCS  p1, 0xF, R1,R3,c2
```

代码清单28.2:ARM架构(Thumb 模式)不正确的反汇编代码示例

```
    LSRS    R3, R6, #0x12
    LDRH    R1, [R7,#0x2C]
    SUBS    R0, #0x55 ; 'U'
    ADR     R1, loc_3C
    LDR     R2, [SP,#0x218]
    CMP     R4, #0x86
    SXTB    R7, R4
    LDR     R4, [R1,#0x4C]
    STR     R4, [R4,R2]
    STR     R0, [R6,#0x20]
    BGT     0xFFFFFF72
    LDRH    R7, [R2,#0x34]
    LDRSH   R0, [R2,R4]
    LDRB    R2, [R7,R2]

    DCB 0x17
    DCB 0xED

    STRB    R3, [R1,R1]
    STR     R5, [R0,#0x6C]
    LDMIA   R3, {R0-R5,R7}
    ASRS    R3, R2, #3
    LDR     R4, [SP,#0x2C4]
    SVC     0xB5
    LDR     R6, [R1,#0x40]
    LDR     R5, =0xB2C5CA32
    STMIA   R6, {R1-R4,R6}
    LDR     R1, [R3,#0x3C]
    STR     R1, [R5,#0x60]
    BCC     0xFFFFFF70
    LDR     R4, [SP,#0x1D4]
    STR     R5, [R5,#0x40]
    ORRS    R5, R7

loc_3C ; DATA XREF: ROM:00000006
    B       0xFFFFFF98

    ASRS    R4, R1, #0x1E
    ADDS    R1, R3, R0
    STRH    R7, [R7,#0x30]
    LDR     R3, [SP,#0x230]
    CBZ     R6, loc_90
    MOVS    R4, R2
    LSRS    R3, R4, #0x17
    STMIA   R6!, {R2,R4,R5}
    ADDS    R6, #0x42 ; 'B'
    ADD     R2, SP, #0x180
    SUBS    R5, R0, R6
    BCC     loc_B0
    ADD     R2, SP, #0x160
```

```
    LSLS    R5, R0, #0x1A
    CMP     R7, #0x45
    LDR     R4, [R4,R5]

    DCB 0x2F ; /
    DCB 0xF4

    B       0xFFFFFD18

    ADD     R4, SP, #0x2C0
    LDR     R1, [SP,#0x14C]
    CMP     R4, #0xEE

    DCB  0xA
    DCB 0xFB

    STRH    R7, [R5,#0xA]
    LDR     R3, loc_78

    DCB 0xBE ; -
    DCB 0xFC

    MOVS    R5, #0x96

    DCB 0x4F ; O
    DCB 0xEE

    B       0xFFFFFAE6

    ADD     R3, SP, #0x110

loc_78 ; DATA XREF: ROM:0000006C
    STR     R1, [R3,R6]
    LDMIA   R3!, {R2,R5-R7}
    LDRB    R2, [R4,R2]
    ASRS    R4, R0, #0x13
    BKPT    0xD1
    ADDS    R5, R0, R6
    STR     R5, [R3,#0x58]
```

代码清单28.2:MIPS架构(小端序)不正确的反汇编代码示例

```
lw      $t9, 0xCB3($t5)
sb      $t5, 0x3855($t0)
sltiu   $a2, $a0, -0x657A
ldr     $t4, -0x4D99($a2)
daddi   $s0, $s1, 0x50A4
lw      $s7, -0x2353($s4)
bgtzl   $a1, 0x17C5C

.byte 0x17
```

```
.byte 0xED
.byte 0x4B  # K
.byte 0x54  # T

lwc2    $31, 0x66C5($sp)
lwu     $s1, 0x10D3($a1)
ldr     $t6, -0x204B($zero)
lwc1    $f30, 0x4DBE($s2)
daddiu  $t1, $s1, 0x6BD9
lwu     $s5, -0x2C64($v1)
cop0    0x13D642D
bne     $gp, $t4, 0xFFFF9EF0
lh      $ra, 0x1819($s1)
sdl     $fp, -0x6474($t8)
jal     0x78C0050
ori     $v0, $s2, 0xC634
blez    $gp, 0xFFFEA9D4
swl     $t8, -0x2CD4($s2)
sltiu   $a1, $k0, 0x685
sdc1    $f15, 0x5964($at)
sw      $s0, -0x19A6($a1)
sltiu   $t6, $a3, -0x66AD
lb      $t7, -0x4F6($t3)
sd      $fp, 0x4B02($a1)

.byte 0x96
.byte 0x25  # %
.byte 0x4F  # O
.byte 0xEE

swl     $a0, -0x1AC9($k0)
lwc2    $4, 0x5199($ra)
bne     $a2, $a0, 0x17308

.byte 0xD1
.byte 0xBE
.byte 0x85
.byte 0x19

swc2    $8, 0x659D($a2)
swc1    $f8, -0x2691($s6)
sltiu   $s6, $t4, -0x2691
sh      $t9, -0x7992($t4)
bne     $v0, $t0, 0x163A4
sltiu   $a3, $t2, -0x60DF
lbu     $v0, -0x11A5($v1)
pref    0x1B, 0x362($gp)
pref    7, 0x3173($sp)
blez    $t1, 0xB678
swc1    $f3, flt_CE4($zero)
pref    0x11, -0x704D($t4)
ori     $k1, $s2, 0x1F67
swr     $s6, 0x7533($sp)
```

```
swc2    $15, -0x67F4($k0)
ldl     $s3, 0xF2($t7)
bne     $s7, $a3, 0xFFFE973C
sh      $s1, -0x11AA($a2)
bnel    $a1, $t6, 0xFFFE566C
sdr     $s1, -0x4D65($zero)
sd      $s2, -0x24D7($t8)
scd     $s4, 0x5C8D($t7)

.byte 0xA2
.byte 0xE8
.byte 0x5C  # \
.byte 0xED

bgtz    $t3, 0x189A0
sd      $t6, 0x5A2F($t9)
sdc2    $10, 0x3223($k1)
sb      $s3, 0x5744($t9)
lwr     $a2, 0x2C48($a0)
beql    $fp, $s2, 0xFFFF3258
```

同样重要的是要记住，巧妙地运用解压缩和解密技术(包括自修改)，可能看起来像是一段不正确的反汇编代码，但是，它是能够正确运行的(注1)。

注1: 一段代码在经过压缩或者加密之后，他的机器码全都变乱了，因此，反汇编结果得到的是一段错误的反汇编代码。但是经过一段解压缩程序或者解密程序处理之后，它就能够还原出原来的机器码，因此反汇编出来的代码和运行结果都是正确的。

第五十章

花指令

花指令是企图隐藏掉不想被逆向工程的代码块(或其它功能)的一种方法。

50.1 文本字符串

我发现在文本字符串使用可能会很有用，程序员意识某字符串不想被逆向工程的时候，可能会试图隐藏掉该字符串，让IDA或者其他十六进制编辑器无法找到。 这里说明一个简单的方法，那就是怎么去构造这样的字符串的实现方式：

```
mov byte ptr [ebx], ’h’
mov byte ptr [ebx+1], ’e’
mov byte ptr [ebx+2], ’l’
mov byte ptr [ebx+3], ’l’
mov byte ptr [ebx+4], ’o’
mov byte ptr [ebx+5], ’ ’
mov byte ptr [ebx+6], ’w’
mov byte ptr [ebx+7], ’o’
mov byte ptr [ebx+8], ’r’
mov byte ptr [ebx+9], ’l’
mov byte ptr [ebx+10], ’d’
```

当两个字符串进行比较的时候看起来是这样：

```
mov ebx, offset username
cmp byte ptr [ebx], ’j’
jnz fail
cmp byte ptr [ebx+1], ’o’
jnz fail
cmp byte ptr [ebx+2], ’h’
jnz fail
cmp byte ptr [ebx+3], ’n’
jnz fail
jz it_is_john
```

在这两种情况下，是不可能通过十六进制编辑器中找到这些字符串的。

顺便提一下，这种方法使得字符串不可能被分配到程序的代码段中。在某些场合可能会用到，比如，在PIC或者在shellcode中。

另一种方法是，我曾经看到用sprintf()构造字符串。

```
sprintf(buf, "%s%c%s%c%s", "hel",'l',"o w",'o',"rld");
```

代码看起来比较怪异，但是做为一个简单的防止逆向工程确实一个有用的方法。 文本字符串也可能存在于加密的形式，那么所有字符串在使用前比较闲将字符串解密了。

50.2 可执行代码

50.2.1 插入垃圾

可执行代码花指令的意思是在真实的代码中插入一些垃圾代码，但是保证原有程序的执行正确。

举个简单的例子：

```
add eax, ebx
mul ecx
```

代码清单29.1：花指令

```
xor esi, 011223344h ; garbage
add esi, eax ; garbage
add eax, ebx
mov edx, eax ; garbage
shl edx, 4 ; garbage
mul ecx
xor esi, ecx ; garbage
```

这里的花指令使用原程序代码中没有使用的寄存器(ESI和EDX)。无论如何，增加花指令之后，原有的汇编代码变得更为枯涩难懂，从而达到不轻易被逆向工程的效果。

50.2.2 替换与原有指令等价的指令

```
mov op1, op2可以替换为 push op2/pop op1这两条指令。
jmp label可以替换为 push label/ret这两条指令，IDA将不会显示被引用的labe
l。
call label可以替换为push label_after_call_instruction/push label/r
ef这三条指令。
push op可以替换为 sub esp, 4(或者8)/mov [esp], op这两条指令。
```

50.2.3 绝对被执行的代码与绝对不被执行的代码

如果开发人员肯定ESI寄存器始终为0：

```
    mov esi, 1
    ... ; some code not touching ESI
    dec esi
    ... ; some code not touching ESI
    cmp esi, 0
    jz real_code
    ;fakeluggage
real_code:
```

逆向工程需要一段时间才能够执行到real_code。这也被称为opaque predicate。另一个例子(同上，假设可以肯定ESI寄存器始终为0):

```
add eax, ebx ; real code
mul ecx ; real code
add eax, esi ; opaque predicate. XOR, AND or SHL, etc, can be he
re instead of ADD.
```

50.2.4打乱执行流程

举个例子，比如执行下面这三条指令：

```
instruction 1
instruction 2
instruction 3
```

可以被替换为：

```
begin:
    jmp ins1_label
ins2_label:
    instruction 2
    jmp ins3_label
ins3_label:
    instruction 3
    jmp exit
ins1_label:
    instruction 1
    jmp ins2_label
exit:
```

50.2.4使用间接指针

```
dummy_data1 db 100h dup (0)
message1 db 'hello world',0

dummy_data2 db 200h dup (0)
message2 db 'another message',0

func proc
    ...
    mov eax, offset dummy_data1 ; PE or ELF reloc here
    add eax, 100h
    push eax
    call dump_string
    ...
    mov eax, offset dummy_data2 ; PE or ELF reloc here
    add eax, 200h
    push eax
    call dump_string
    ...
func endp
```

IDA仅会显示dummy_data1和dummy_data2的引用，但无法引导到文本字符串，全局变量甚至是函数的访问方式都可能使用这种方法以达到混淆代码的目地。

50.3 虚拟机/伪代码

程序员可能写一个PL或者ISA来解释程序(例如Visual Basic 5.0与之前的版本, .NET, Java machine)。这使得逆向工程不得不花费更多的时间去了解这些语言它们的所有ISP指令详细信息。更有甚者，他们可能需要编写其中某些语言的反汇编器。

50.4 其它

我为TCC(Tiny C compiler)添加一个产生花指令功能的补丁：
http://blog.yurichev.com/node/58。

50.5练习

- http://challenges.re/29

第五十一章

C++

31.1 类

51.1.1 简单的例子

在程序内部，C++类的表示基本和结构体一样。让我们试试这个有2个变量，2个构造函数和1个方法的类。

```
#include <stdio.h>
class c
{
private:
    int v1;
    int v2;
public:
    c() // default ctor
    {
        v1=667;
        v2=999;
    };
    c(int a, int b) // ctor
    {
        v1=a;
        v2=b;
    };
    void dump()
    {
        printf ("%d; %d", v1, v2);
    };
};

int main()
{
    class c c1;
    class c c2(5,6);
    c1.dump();
    c2.dump();
    return 0;
};
```

MSVC-X86

这里可以看到main（）函数是如何被翻译成汇编代码的：

```
_c2$ = -16 ; size = 8
_c1$ = -8 ; size = 8
_main PROC
push ebp
mov ebp, esp
sub esp, 16 ; 00000010H
lea ecx, DWORD PTR _c1$[ebp]
call ??0c@@QAE@XZ ; c::c
push 6
push 5
lea ecx, DWORD PTR _c2$[ebp]
call ??0c@@QAE@HH@Z ; c::c
lea ecx, DWORD PTR _c1$[ebp]
call ?dump@c@@QAEXXZ ; c::dump
lea ecx, DWORD PTR _c2$[ebp]
call ?dump@c@@QAEXXZ ; c::dump
xor eax, eax
mov esp, ebp
pop ebp
ret 0
_main ENDP
```

所以，发生什么了。对每个对象来说（而不是类c），会分配8个字节。这正好是2个变量存储所需的大小。 对c1来说一个默认的无参数构造函数??0c@@QAE@XZ会被调用。对c2来说另一个??0c@@QAE@HH@Z会被调用，有两个数字会被作为参数传递。 指向对象的指针（C++术语的“this”）会被通过ECX寄存器传递。这被叫做thiscall（31.1.1）--这是一个指向对象的指针传递方式。 MSVC使用ECX来传递它。无需说明的是，它并不是一个标准化的方法，其他编译器可能用其他方法，例如通过第一个函数参数，比如GCC就是这么做的。 为什么函数的名字这么奇怪？这是因为名字打碎方式的缘故。 C++类可能有多个同名的重载函数，因此，不同的类也可能有相同的函数名。 名字打碎可以把类的类名+函数名+参数类型编码到一个字符串里面，然后它就会被用作内部名称。这完全是因为编译器和DLL OS加载器都 不知道C++或者面向对象的缘故。 Dump()函数在之后被调用了2次。 让我们看看构造函数的代码。

```
_this$ = -4 ; size = 4
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
push ebp
mov ebp, esp
push ecx
mov DWORD PTR _this$[ebp], ecx
mov eax, DWORD PTR _this$[ebp]
mov DWORD PTR [eax], 667 ; 0000029bH
mov ecx, DWORD PTR _this$[ebp]
mov DWORD PTR [ecx+4], 999 ; 000003e7H
mov eax, DWORD PTR _this$[ebp]
mov esp, ebp
pop ebp
ret 0


??0c@@QAE@XZ ENDP ; c::c
_this$ = -4 ; size = 4
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
push ebp
mov ebp, esp
push ecx
mov DWORD PTR _this$[ebp], ecx
mov eax, DWORD PTR _this$[ebp]
mov ecx, DWORD PTR _a$[ebp]
mov DWORD PTR [eax], ecx
mov edx, DWORD PTR _this$[ebp]
mov eax, DWORD PTR _b$[ebp]
mov DWORD PTR [edx+4], eax
mov eax, DWORD PTR _this$[ebp]
mov esp, ebp
pop ebp
ret 8
??0c@@QAE@HH@Z ENDP ; c::c
```

构造函数只是函数，它们会使用ECX中存储的指向结构体的指针，然后把指针指向自己的本地变量，但是，这个操作并不是必须的。 对C++标准来说我们知道构造函数不应该返回任何值。事实上，构造函数会返回指向新创建对象的指针，比如“this”。 现在看看dump（）函数：

```
_this$ = -4 ; size = 4
?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
push ebp
mov ebp, esp
push ecx
mov DWORD PTR _this$[ebp], ecx
mov eax, DWORD PTR _this$[ebp]
mov ecx, DWORD PTR [eax+4]
push ecx
mov edx, DWORD PTR _this$[ebp]
mov eax, DWORD PTR [edx]
push eax
push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
call _printf
add esp, 12 ; 0000000cH
mov esp, ebp
pop ebp
ret 0
?dump@c@@QAEXXZ ENDP ; c::dump
```

简单的可以：dump()会把带有2个int的结构体传给ecx，然后从他里面取出2个值，然后传给printf()。 如果使用/Ox优化，代码会更短。

```
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
mov eax, ecx
mov DWORD PTR [eax], 667 ; 0000029bH
mov DWORD PTR [eax+4], 999 ; 000003e7H
ret 0
??0c@@QAE@XZ ENDP ; c::c
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
mov edx, DWORD PTR _b$[esp-4]
mov eax, ecx
mov ecx, DWORD PTR _a$[esp-4]
mov DWORD PTR [eax], ecx
mov DWORD PTR [eax+4], edx
ret 8
??0c@@QAE@HH@Z ENDP ; c::c
?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+4]
mov ecx, DWORD PTR [ecx]
push eax
push ecx
push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
call _printf
add esp, 12 ; 0000000cH
ret 0
?dump@c@@QAEXXZ ENDP ; c::dump
```

还要说的就是栈指针在调用add esp ，x之后并不正确。所以构造函数还需要ret 8来返回，而不是ret。 这是因为这儿调用方式是thiscall（31.1.1）,这个方法会使用栈来传递参数，和stdcall对比（47.2）来看，他将为被调用者维护正确的栈，而不是调用者。Ret x指令会额外的给esp加上x，然后会把控制流交还给调用者函数。 调用转换见47章。 还有需要注意的是，编译器会决定什么时候调用构造函数什么时候调用析构函数，但是我们从c++语言基础里面已经知道调用时机了。

MSVC-x86-64

像我们已经知道的那样，x86-64中前4个函数参数是通过RCX/RDX/R8/R9寄存器传递的，剩余的通过栈传递。但是this是用RCX传递的 ，而第一个函数参数是从RDX开始传递的。我们可以通过c(int a, int b)这个函数看出来。

```
; void dump()
?dump@c@@QEAAXXZ PROC ; c::dump
mov r8d, DWORD PTR [rcx+4]
mov edx, DWORD PTR [rcx]
lea rcx, OFFSET FLAT:??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@ ;
’%d; %d’
jmp printf
?dump@c@@QEAAXXZ ENDP ; c::dump
; c(int a, int b)
??0c@@QEAA@HH@Z PROC ; c::c
mov DWORD PTR [rcx], edx ; 1st argument: a
mov DWORD PTR [rcx+4], r8d ; 2nd argument: b
mov rax, rcx
ret 0
??0c@@QEAA@HH@Z ENDP ; c::c
; default ctor
??0c@@QEAA@XZ PROC ; c::c
mov DWORD PTR [rcx], 667 ; 0000029bH
mov DWORD PTR [rcx+4], 999 ; 000003e7H
mov rax, rcx
ret 0
??0c@@QEAA@XZ ENDP ; c::c
```

X64中，Int数据类型依然是32位的。所以这里也使用了32位寄存器部分。 我们还可以看到dump()里的JMP printf，而不是RET，这个技巧我们已经在11.1.1里面见过了。

GCC-x86

几乎和GCC4.4.1一样的结果，除了几个例外。

```
public main
main proc near ; DATA XREF: _start+17
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_18 = dword ptr -18h
var_10 = dword ptr -10h
var_8 = dword ptr -8
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 20h
lea eax, [esp+20h+var_8]
mov [esp+20h+var_20], eax
call _ZN1cC1Ev
mov [esp+20h+var_18], 6
mov [esp+20h+var_1C], 5
lea eax, [esp+20h+var_10]
mov [esp+20h+var_20], eax
call _ZN1cC1Eii
lea eax, [esp+20h+var_8]
mov [esp+20h+var_20], eax
call _ZN1c4dumpEv
lea eax, [esp+20h+var_10]
mov [esp+20h+var_20], eax
call _ZN1c4dumpEv
mov eax, 0
leave
retn
main endp
```

我们可以看到另一个命名破碎模式，这个GNU特殊的模式可以看到指向对象的this时针其实是作为函数的第一个参数被传入的，当然，这个对程序员来说是透明的。第一个构造函数：

```
public _ZN1cC1Ev ; weak
_ZN1cC1Ev proc near ; CODE XREF: main+10
arg_0 = dword ptr 8
push ebp
mov ebp, esp
mov eax, [ebp+arg_0]
mov dword ptr [eax], 667
mov eax, [ebp+arg_0]
mov dword ptr [eax+4], 999
pop ebp
retn
_ZN1cC1Ev endp
```

他所做的无非就是使用第一个传来的参数写入两个数字。 第二个构造函数：

```
public _ZN1cC1Eii
_ZN1cC1Eii proc near
arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h
push ebp
mov ebp, esp
mov eax, [ebp+arg_0]
mov edx, [ebp+arg_4]
mov [eax], edx
mov eax, [ebp+arg_0]
mov edx, [ebp+arg_8]
mov [eax+4], edx
pop ebp
retn
_ZN1cC1Eii endp
```

这是个函数，原型类似于：

```
void ZN1cC1Eii (int *obj, int a, int b)
{
    *obj=a;
    *(obj+1)=b;
};
```

这是完全可以预测到的，现在看看dump（）：

```
public _ZN1c4dumpEv
_ZN1c4dumpEv proc near
var_18 = dword ptr -18h
var_14 = dword ptr -14h
var_10 = dword ptr -10h
arg_0 = dword ptr 8
push ebp
mov ebp, esp
sub esp, 18h
mov eax, [ebp+arg_0]
mov edx, [eax+4]
mov eax, [ebp+arg_0]
mov eax, [eax]
mov [esp+18h+var_10], edx
mov [esp+18h+var_14], eax
mov [esp+18h+var_18], offset aDD ; "%d; %d
"
call _printf
leave
retn
_ZN1c4dumpEv endp
```

在这个函数的内部表达中有一个单独的参数，被用作指向当前对象，也即this。 因此，如果从这些简单的例子来看，MSVC和GCC的区别也就只有函数名编码的区别和传入this指针的区别（ECX寄存器或通过第一个参数）。

GCC-X86-64

前6个参数，会通过RDI/RSI/RDX/RCX/R8/R9[21章]的顺序传递，this指针会通过第一个RDI来传递，我们可以接着看到。 Int数据类型也是一个32位的数据，JMP替换RET的技巧这里也用到了。

```
; default ctor
_ZN1cC2Ev:
mov DWORD PTR [rdi], 667
mov DWORD PTR [rdi+4], 999
ret
; c(int a, int b)
_ZN1cC2Eii:
mov DWORD PTR [rdi], esi
mov DWORD PTR [rdi+4], edx
ret
; dump()
_ZN1c4dumpEv:
mov edx, DWORD PTR [rdi+4]
mov esi, DWORD PTR [rdi]
xor eax, eax
mov edi, OFFSET FLAT:.LC0 ; "%d; %d
"
jmp printf
```

51.1.2 类继承

可以说关于类继承就是我们已经研究了的这个结构体，但是它现在扩展成类了。 让我们看个简单的例子：

```
#include <stdio.h>
class object
{
public:
int color;
object() { };
object (int color) { this->color=color; };
void print_color() { printf ("color=%d
", color); };
};
class box : public object
{
private:
int width, height, depth;
public:
```

```
box(int color, int width, int height, int depth)
{
this->color=color;
this->width=width;
this->height=height;
this->depth=depth;
};
void dump()
{
printf ("this is box. color=%d, width=%d, height=%d, depth=%d
", color, width,
height, depth);
};
};
class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    };
void dump()
{
    printf ("this is sphere. color=%d, radius=%d", color, radius
);};
};
int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);
    b.print_color();
    s.print_color();
    b.dump();
    s.dump();
    return 0;
};
```

让我们观察一下生成的dump()的代码和object::print_color()，让我们看看结构体对象的内存输出（作为32位代码） 所以，dump()方法其实是对应了好几个类，下面代码由MSVC 2008生成（/Ox+/Ob0） 优化的MSVC 2008 /Ob0

```
??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ;
 'string'
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx]
push eax
; 'color=%d', 0aH, 00H
push OFFSET ??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@
call _printf
add esp, 8
ret 0
?print_color@object@@QAEXXZ ENDP ; object::print_color
```

优化的MSVC2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+12]
mov edx, DWORD PTR [ecx+8]
push eax
mov eax, DWORD PTR [ecx+4]
mov ecx, DWORD PTR [ecx]
push edx
push eax
push ecx
; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 0
0H ; 'string'
push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0
?5width?$DN?$CFd?0@
call _printf
add esp, 20 ; 00000014H
ret 0
?dump@box@@QAEXXZ ENDP ; box::dump

?dump@sphere@@QAEXXZ PROC ; sphere::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+4]
mov ecx, DWORD PTR [ecx]
push eax
push ecx
; 'this is sphere. color=%d, radius=%d', 0aH, 00H
push OFFSET ??_C@_0CF@EFEDJLDC@this?5is?5sphere?4?5color?$DN?$CF
d?0?5radius@
call _printf
add esp, 12 ; 0000000cH
ret 0
?dump@sphere@@QAEXXZ ENDP ; sphere::dump
```

所以，这就是他的内存暑促后：（基类对象）

offset	description
+0x0	int color

继承的对象 Box：

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

Sphere:

offset	description
+0x0	int color
+0x4	int radius

让我们看看main()函数体：

```
PUBLIC _main
_TEXT SEGMENT
_s$ = -24 ; size = 8
_b$ = -16 ; size = 16
_main PROC
sub esp, 24 ; 00000018H
push 30 ; 0000001eH
push 20 ; 00000014H
push 10 ; 0000000aH
push 1
lea ecx, DWORD PTR _b$[esp+40]
call ??0box@@QAE@HHHH@Z ; box::box
push 40 ; 00000028H
push 2
lea ecx, DWORD PTR _s$[esp+32]
call ??0sphere@@QAE@HH@Z ; sphere::sphere
lea ecx, DWORD PTR _b$[esp+24]
call ?print_color@object@@QAEXXZ ; object::print_color
lea ecx, DWORD PTR _s$[esp+24]
call ?print_color@object@@QAEXXZ ; object::print_color
lea ecx, DWORD PTR _b$[esp+24]
call ?dump@box@@QAEXXZ ; box::dump
lea ecx, DWORD PTR _s$[esp+24]
call ?dump@sphere@@QAEXXZ ; sphere::dump
xor eax, eax
add esp, 24 ; 00000018H
ret 0
_main ENDP
```

继承的类必须永远将它们的范围添加到基类的范围中，所以这样可以让基类的方法对其范围生效。当object::print_color()方法被调用时，会有一个指针指向box对象和sphere对象会被传递进去，它就是“this”。它可以和这些对象简单的互动，因为color域指向的永远是固定的地址（+0x00偏移）。可以说，object::print_color()方

法对于输入对象类型来说是不可知的，如果你创建一个继承类，例如继承了box类编译器会自动在depth域之后加上新域，而把box的类域固定在一个固定的位置。 因此，box::dump()方法会在访问color/width/height/depths的时候顺利工作，因为地址的固定，它会很容易的知道偏移。 GCC生成的代码基本一样，只有一个不一样的就是this的传递，就像之前说的一样，它是作为第一个参数传递的，而不是通过ECX传递的。

51..13 封装

封装是一个把数据装在类的private域里面的动作，这样会让它们只能从类的内部被访问到，而从外面访问不到。 但是，生成的代码里面是否有什么东西指示一个变量是private呢？ 没有，让我们看看简单的例子：

```
#include <stdio.h>
class box
{
private:
int color, width, height, depth;
public:
box(int color, int width, int height, int depth)
{
this->color=color;
this->width=width;
this->height=height;
this->depth=depth;
};
void dump()
{
printf ("this is box. color=%d, width=%d, height=%d, depth=%d
", color, width,
height, depth);
};
};
```

在MSVC 2008+/Ox和/Ob0选项，然后看看box::dump()代码：

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+12]
mov edx, DWORD PTR [ecx+8]
push eax
mov eax, DWORD PTR [ecx+4]
mov ecx, DWORD PTR [ecx]
push edx
push eax
push ecx
; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 0
0H
push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0
?5width?$DN?$CFd?0@
call _printf
add esp, 20 ; 00000014H
ret 0
?dump@box@@QAEXXZ ENDP ; box::dump
```

这就是类的内存分布：

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

所有域都不允许其他类的访问，但是，我们知道这个存放方式之后是否可以修改这些域？所以我加了hack_oop_encapsulation()函数，假设他有这个代码，当然我们没有编译：

```
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // that code can't be compiled: "error C2248: 'b
ox::width' : cannot access
    private member declared in class 'box'"
};
```

还有，如果要转换box的类型，把它从指针转为int数组，然后如果我们能修改这些数字，那么我们就成功了。

```
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(
o);
    ptr_to_object[1]=123;
};
```

这个函数的代码非常简单，剋说函数指示把指针指向这些int，然后把123写入第二个int：

```
?hack_oop_encapsulation@@YAXPAVbox@@@Z PROC ; hack_oop_encapsula
tion
mov eax, DWORD PTR _o$[esp-4]
mov DWORD PTR [eax+4], 123 ; 0000007bH
ret 0
?hack_oop_encapsulation@@YAXPAVbox@@@Z ENDP ; hack_oop_encapsula
tion
```

看看它是怎么工作的：

```
int main()
{
    box b(1, 10, 20, 30);
    b.dump();
    hack_oop_encapsulation(&b);
    b.dump();
    return 0;
};
```

运行后：

```
this is box. color=1, width=10, height=20, depth=30
this is box. color=1, width=123, height=20, depth=30
```

可以看到，private只是在编译阶段被保护了，c++编译器不会允许其他代码修改private域下的内容，但是如果用一些技巧，就可以修改private的值。

51.1.4 多重继承

多重继承是一个类的创建，这个类会从2个或多个类里面继承函数和成员。 看一个简单的例子：

```
#include <stdio.h>
class box
{
public:
    int width, height, depth;
    box() { };
    box(int width, int height, int depth)
    {
        this->width=width;
        this->height=height;
        this->depth=depth;
```

```
    };
    void dump()
    {
        printf ("this is box. width=%d, height=%d, depth=%d", wi
dth, height, depth);
    };
    int get_volume()
    {
        return width * height * depth;
    };
};
class solid_object
{
public:
    int density;
    solid_object() { };
    solid_object(int density)
    {
        this->density=density;
    };
    int get_density()
    {
        return density;
    };
    void dump()
    {
        printf ("this is solid_object. density=%d", density);
    };
};
class solid_box: box, solid_object
{
public:
    solid_box (int width, int height, int depth, int density)
    {
        this->width=width;
        this->height=height;
        this->depth=depth;
        this->density=density;
    };
    void dump()
    {
        printf ("this is solid_box. width=%d, height=%d, depth=%
d, density=%d", width, height, depth, density);
    };
    int get_weight() { return get_volume() * get_density(); };
};
int main()
{
    box b(10, 20, 30);
    solid_object so(100);
    solid_box sb(10, 20, 30, 3);
    b.dump();
    so.dump();
```

```
    sb.dump();
    printf ("%d", sb.get_weight());
    return 0;
};
```

让我们在MSVC 2008中用/Ox和/Ob0选项来编译，然后看看box::dump()、solid_object::dump()和solid_box::dump()的函数代码：

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+8]
mov edx, DWORD PTR [ecx+4]
push eax
mov eax, DWORD PTR [ecx]
push edx
push eax
; 'this is box. width=%d, height=%d, depth=%d', 0aH, 00H
push OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?5width?$DN?$CFd?0
?5height?$DN?$CFd@
call _printf
add esp, 16 ; 00000010H
ret 0
?dump@box@@QAEXXZ ENDP ; box::dump

?dump@solid_object@@QAEXXZ PROC ; solid_object::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx]
push eax
; 'this is solid_object. density=%d', 0aH
push OFFSET ??_C@_0CC@KICFJINL@this?5is?5solid_object?4?5density
?$DN?$CFd@
call _printf
add esp, 8
ret 0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump

?dump@solid_box@@QAEXXZ PROC ; solid_box::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+12]
mov edx, DWORD PTR [ecx+8]
push eax
mov eax, DWORD PTR [ecx+4]
mov ecx, DWORD PTR [ecx]
push edx
push eax
push ecx
; 'this is solid_box. width=%d, height=%d, depth=%d, density=%d'
, 0aH
push OFFSET ??_C@_0DO@HNCNIHNN@this?5is?5solid_box?4?5width?$DN?
$CFd?0?5hei@
call _printf
add esp, 20 ; 00000014H
ret 0
?dump@solid_box@@QAEXXZ ENDP ; solid_box::dump
```

所以，这三个类的内存分布是：

Box：

offset	description
+0x0	width
+0x4	height
+0x8	depth

Solid_object：

offset	description
+0x0	density

可以说，solid_box的类内存空间就是它们的组合：

offset	description
+0x0	width
+0x4	height
+0x8	depth
+0xC	density

Box::get_volume()和solid_object::get_density()函数的代码如下：

```
?get_volume@box@@QAEHXZ PROC ; box::get_volume, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+8]
imul eax, DWORD PTR [ecx+4]
imul eax, DWORD PTR [ecx]
ret 0
?get_volume@box@@QAEHXZ ENDP ; box::get_volume

?get_density@solid_object@@QAEHXZ PROC ; solid_object::get_densi
ty, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx]
ret 0
?get_density@solid_object@@QAEHXZ ENDP ; solid_object::get_densi
ty
```

但是solid_box::get_weight()的代码更有趣：

```
?get_weight@solid_box@@QAEHXZ PROC ; solid_box::get_weight, COMD
AT
; _this$ = ecx
push esi
mov esi, ecx
push edi
lea ecx, DWORD PTR [esi+12]
call ?get_density@solid_object@@QAEHXZ ; solid_object::get_densi
ty
mov ecx, esi
mov edi, eax
call ?get_volume@box@@QAEHXZ ; box::get_volume
imul eax, edi
pop edi
pop esi
ret 0
?get_weight@solid_box@@QAEHXZ ENDP ; solid_box::get_weight
```

Get_weight()函数只会调用2个函数，但是对于get_volume()来说，他只是传递指针给this，对get_density()来说，他指示传递指针给this，同时移位12（0xC）字节，然后在solid_box类的内存空间理，solid_object类开始了。 因此，solid_object::get_density()方法相信它正在处理普通的solid_object类，而且box::get_volume类将对它的3个域生效，而且相信这是普通的box类对象。 因此，我们可以说，类的一个对象，是从多个其他类继承阿日来，在内存中代表着组合起来的类，因为它有所有继承来的域。每个继承的方法都会又一个指向对应结构部分的指针来处理。

51.1.5 虚拟方法

还有一个简单的例子：

```
#include <stdio.h>
class object
{
public:
    int color;
    object() { };
    object (int color) { this->color=color; };
    virtual void dump()
    {
        printf ("color=%d", color);
    };
};
class box : public object
{
private:
    int width, height, depth;
public:
    box(int color, int width, int height, int depth)
```

```
    {
        this->color=color;
        this->width=width;
        this->height=height;
        this->depth=depth;
    };
    void dump()
    {
        printf ("this is box. color=%d, width=%d, height=%d, dep
th=%d", color, width,height, depth);
    };
};
class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    };
void dump()
{
    printf ("this is sphere. color=%d, radius=%d", color, radius
);};
};
int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);
    object *o1=&b;
    object *o2=&s;
    o1->dump();
    o2->dump();
    return 0;
};
```

类object有一个虚函数dump()，被box和sphere类继承者替换。 如果在一个并不知道什么类型是什么对象的环境下，就像在main()这个函数里面一样，当一个虚函数dump()被调用的时候，我们还是需要知道它的返回类型的。 让我们在MSVC2008用/Ox 、 /Ob0编译看看main()的函数代码：

```
_s$ = -32 ; size = 12
_b$ = -20 ; size = 20
_main PROC
sub esp, 32 ; 00000020H
push 30 ; 0000001eH
push 20 ; 00000014H
push 10 ; 0000000aH
push 1
lea ecx, DWORD PTR _b$[esp+48]
call ??0box@@QAE@HHHH@Z ; box::box
push 40 ; 00000028H
push 2
lea ecx, DWORD PTR _s$[esp+40]
call ??0sphere@@QAE@HH@Z ; sphere::sphere
mov eax, DWORD PTR _b$[esp+32]
mov edx, DWORD PTR [eax]
lea ecx, DWORD PTR _b$[esp+32]
call edx
mov eax, DWORD PTR _s$[esp+32]
mov edx, DWORD PTR [eax]
lea ecx, DWORD PTR _s$[esp+32]
call edx
xor eax, eax
add esp, 32 ; 00000020H
ret 0
_main ENDP
```

指向dump()函数的指针在这个对象的某处被使用了，那么新函数的地址写到了哪里呢？只有在构造函数中有可能：其他地方都不会被main()调用。 看看类构造函数的代码：

```
??_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@ ; box ‘RTTI Type Desc
riptor’
DD 00H
DB ’.?AVbox@@’, 00H
??_R1A@?0A@EA@box@@8 DD FLAT:??_R0?AVbox@@@8 ; box::‘RTTI Base C
lass Descriptor at
(0,-1,0,64)’
DD 01H
DD 00H
DD 0ffffffffH
DD 00H
DD 040H
DD FLAT:??_R3box@@8
??_R2box@@8 DD FLAT:??_R1A@?0A@EA@box@@8 ; box::‘RTTI Base Class
 Array’
DD FLAT:??_R1A@?0A@EA@object@@8
??_R3box@@8 DD 00H ; box::‘RTTI Class Hierarchy Descriptor’
DD 00H
DD 02H
DD FLAT:??_R2box@@8
??_R4box@@6B@ DD 00H ; box::‘RTTI Complete Object Locator’
DD 00H
DD 00H
DD FLAT:??_R0?AVbox@@@8
DD FLAT:??_R3box@@8
??_7box@@6B@ DD FLAT:??_R4box@@6B@ ; box::‘vftable’
DD FLAT:?dump@box@@UAEXXZ
_color$ = 8 ; size = 4
_width$ = 12 ; size = 4
_height$ = 16 ; size = 4
_depth$ = 20 ; size = 4
??0box@@QAE@HHHH@Z PROC ; box::box, COMDAT
; _this$ = ecx
push esi
mov esi, ecx
call ??0object@@QAE@XZ ; object::object
mov eax, DWORD PTR _color$[esp]
mov ecx, DWORD PTR _width$[esp]
mov edx, DWORD PTR _height$[esp]
mov DWORD PTR [esi+4], eax
mov eax, DWORD PTR _depth$[esp]
mov DWORD PTR [esi+16], eax
mov DWORD PTR [esi], OFFSET ??_7box@@6B@
mov DWORD PTR [esi+8], ecx
mov DWORD PTR [esi+12], edx
mov eax, esi
pop esi
ret 16 ; 00000010H
??0box@@QAE@HHHH@Z ENDP ; box::box
```

我们可以看到一些轻微的内存布局的变化：第一个域是一个指向box::vftable（这个名字由MSVC编译器生成）的指针。 在这个函数表里我们看到了一个指向box::RTTI Complete Object Locator的连接，而且还有一个指向box::dump()函数的。所以这就是被命名的虚函数表和RTTI。虚函数表可以包含所有虚函数体的地址，RTTI表包含类型的信息。另外一提，RTTI表是c++调用dynamic_cast和typeid的结果的枚举表。你可以看到这里函数名是用明文表记的。因此，一个基对象可以调用虚函数object::dump()，然后，会从这个对象的结构里调用这个继承类的函数。枚举这些函数表需要消耗额外的CPU时间，所以可以认为虚函数比普通调用要慢一些。 在GCC生成的代码里，RTTI表的构造有些轻微的不同。

51.2 输出流

51.3 References

51.4 STL

51.4 STL

Internals

MSVC

GCC

一个更加复杂的例子

std::string 作为全局变量

51.4.2 std::list

GCC

MSVC

C++11 std::forward_list

51.4.3 std::vector

51.4.4 std::map and std::set

MSVC

GCC

Rebalancing demo (GCC)

#

负的数组引索

第五十三章

16位Windows

16位windows程序现在很少见了，但是在旧式计算机或者入侵软件狗的时候（58章），我有时候还会遇到这个问题。 16位的windows版本最高到3.11，95(*注：作者笔误写成了Win96)/98/ME也支持16位代码，他们同时也是一个Windows NT家族的32位版本。64位版本的Windows NT家族完全不支持16位程序。 代码类似于MS-DOS代码。 执行文件并不是MZ式或者PE文件，而是NE式（所谓的"New Executable"，新执行程序）。 所有的例子都由OpenWatcom 1.9编译器编译，使用这些参数：

```
Wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows example.c
```

53.1 例子#1

```
#include <windows.h>
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    MessageBeep(MB_ICONEXCLAMATION);
    return 0;
};
```

```
WinMain proc near
    push bp
    mov bp, sp
    mov ax, 30h ; ’0’ ; MB_ICONEXCLAMATION constant
    push ax
    call MESSAGEBEEP
    xor ax, ax ; return 0
    pop bp
    retn 0Ah
WinMain endp
```

到现在为止，看起来都很简单。

53.2 例子#2

```
#include <windows.h>
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL)
;
    return 0;
};
```

```
WinMain proc near
        push bp
        mov bp, sp
        xor ax, ax ; NULL
        push ax
        push ds
        mov ax, offset aHelloWorld ; 0x18. "hello, world"
        push ax
        push ds
        mov ax, offset aCaption ; 0x10. "caption"
        push ax
        mov ax, 3 ; MB_YESNOCANCEL
        push ax
        call MESSAGEBOX
        xor ax, ax ; return 0
        pop bp
        retn 0Ah
WinMain endp
dseg02:0010 aCaption db 'caption',0
dseg02:0018 aHelloWorld db 'hello, world',0
```

有两个重要的信息：PASCAL调用转换表明先传递最后的参数（MB_YESNOCANCEL），然后才是第一个参数NULL。这个调用也表明了调用者恢复栈指针：因为RETN有一个0Ah的参数，这个意味着栈指针将在函数退出时上移10个字节。 指针按对传递：一组数据先传递，指针就在这组数据里面。例子这里只有一组数据，所以DS永远指向可执行文件的data段。

53.3 例子#3

```
#include <windows.h>
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    int result=MessageBox (NULL, "hello, world", "caption", MB_Y
ESNOCANCEL);
    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK
);
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)
        MessageBox (NULL, "you pressed no", "caption", MB_OK);
    return 0;
};
```

```
WinMain proc near
        push bp
        mov bp, sp
        xor ax, ax ; NULL
        push ax
        push ds
        mov ax, offset aHelloWorld ; "hello, world"
        push ax
        push ds
        mov ax, offset aCaption ; "caption"
        push ax
        mov ax, 3 ; MB_YESNOCANCEL
        push ax
        call MESSAGEBOX
        cmp ax, 2 ; IDCANCEL
        jnz short loc_2F
        xor ax, ax
        push ax
        push ds
        mov ax, offset aYouPressedCanc ; "you pressed cancel"
        jmp short loc_49
        ; -------------------------------------------------------
---------------------
        loc_2F:
        cmp ax, 6 ; IDYES
        jnz short loc_3D
        xor ax, ax
        push ax
        push ds
        mov ax, offset aYouPressedYes ; "you pressed yes"
        jmp short loc_49
        ; -------------------------------------------------------
---------------------
        loc_3D:
```

```
        cmp ax, 7 ; IDNO
        jnz short loc_57
        xor ax, ax
        push ax
        push ds
        mov ax, offset aYouPressedNo ; "you pressed no"
        loc_49:
        push ax
        push ds
        mov ax, offset aCaption ; "caption"
        push ax
        xor ax, ax
        push ax
        call MESSAGEBOX
        loc_57:
        xor ax, ax
        pop bp
        retn 0Ah
WinMain endp
```

就是前一节的扩展而已。

53.4 例子#4

```
#include <windows.h>
int PASCAL func1 (int a, int b, int c)
{
    return a*b+c;
};
long PASCAL func2 (long a, long b, long c)
{
    return a*b+c;
};
long PASCAL func3 (long a, long b, long c, int d)
{
    return a*b+c-d;
};

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    func1 (123, 456, 789);
    func2 (600000, 700000, 800000);
    func3 (600000, 700000, 800000, 123);
    return 0;
};
```

```
func1 proc near
```

```
    c = word ptr 4
    b = word ptr 6
    a = word ptr 8
    push bp
    mov bp, sp
    mov ax, [bp+a]
    imul [bp+b]
    add ax, [bp+c]
    pop bp
    retn 6
func1 endp
func2 proc near
    arg_0 = word ptr 4
    arg_2 = word ptr 6
    arg_4 = word ptr 8
    arg_6 = word ptr 0Ah
    arg_8 = word ptr 0Ch
    arg_A = word ptr 0Eh
    push bp
    mov bp, sp
    mov ax, [bp+arg_8]
    mov dx, [bp+arg_A]
    mov bx, [bp+arg_4]
    mov cx, [bp+arg_6]
    call sub_B2 ; long 32-bit multiplication
    add ax, [bp+arg_0]
    adc dx, [bp+arg_2]
    pop bp
    retn 12
func2 endp
func3 proc near
    arg_0 = word ptr 4
    arg_2 = word ptr 6
    arg_4 = word ptr 8
    arg_6 = word ptr 0Ah
    arg_8 = word ptr 0Ch
    arg_A = word ptr 0Eh
    arg_C = word ptr 10h
    push bp
    mov bp, sp
    mov ax, [bp+arg_A]
    mov dx, [bp+arg_C]
    mov bx, [bp+arg_6]
    mov cx, [bp+arg_8]
    call sub_B2 ; long 32-bit multiplication
    mov cx, [bp+arg_2]
    add cx, ax
    mov bx, [bp+arg_4]
    adc bx, dx ; BX=high part, CX=low part
    mov ax, [bp+arg_0]
    cwd ; AX=low part d, DX=high part d
    sub cx, ax
    mov ax, cx
```

```
    sbb bx, dx
    mov dx, bx
    pop bp
    retn 14
    func3 endp
    WinMain proc near
    push bp
    mov bp, sp
    mov ax, 123
    push ax
    mov ax, 456
    push ax
    mov ax, 789
    push ax
    call func1
    mov ax, 9 ; high part of 600000
    push ax
    mov ax, 27C0h ; low part of 600000
    push ax
    mov ax, 0Ah ; high part of 700000
    push ax
    mov ax, 0AE60h ; low part of 700000
    push ax
    mov ax, 0Ch ; high part of 800000
    push ax
    mov ax, 3500h ; low part of 800000
    push ax
    call func2
    mov ax, 9 ; high part of 600000
    push ax
    mov ax, 27C0h ; low part of 600000
    push ax
    mov ax, 0Ah ; high part of 700000
    push ax
    mov ax, 0AE60h ; low part of 700000
    push ax
    mov ax, 0Ch ; high part of 800000
    push ax
    mov ax, 3500h ; low part of 800000
    push ax
    mov ax, 7Bh ; 123
    push ax
    call func3
    xor ax, ax ; return 0
    pop bp
    retn 0Ah
WinMain endp
```

32位的值（long数据类型代表32位，int代表16位数据）在16位模式下（MSDOS和win16）都会按对传递，就像64位数据在32位环境下使用的方式一样（21章）。

Sub_B2在这里是一个编译器生成的库函数，他的作用是“long乘法”，例如两个32位类型想成，其他的编译器函数列在了附录E, D.中。 ADD/ADC指令对用来相加两个值：ADD将设置/清空CF进位标识，ADC将会使用它。 SUB/SBB将会做减法，SUB会设置/清空CF标识位，SBB将会使用它。 32位值按照DX:AX寄存器对返回。常数同样在WinMain()中按照值对的方式传递。 Int类型的123常量首先被转为32位的值，使用的是CWD指令。

53.5 例子#5

```
#include <windows.h>
int PASCAL string_compare (char *s1, char *s2)
{
    while (1)
    {
        if (*s1!=*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    };
};

int PASCAL string_compare_far (char far *s1, char far *s2)
{
    while (1)
    {
        if (*s1!=*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    };
};
void PASCAL remove_digits (char *s)
{
    while (*s)
    {
        if (*s>='0' && *s<='9')
            *s='-';
        s++;
    };
};

char str[]="hello 1234 world";
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    string_compare ("asd", "def");
    string_compare_far ("asd", "def");
    remove_digits (str);
    MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
    return 0;
};
```

```
string_compare proc near
arg_0 = word ptr 4
arg_2 = word ptr 6
```

```
push bp
mov bp, sp
push si
mov si, [bp+arg_0]
mov bx, [bp+arg_2]
loc_12: ; CODE XREF: string_compare+21j
mov al, [bx]
cmp al, [si]
jz short loc_1C
xor ax, ax
jmp short loc_2B
; ---------------------------------------------------------------
-------------
loc_1C: ; CODE XREF: string_compare+Ej
test al, al
jz short loc_22
jnz short loc_27
loc_22: ; CODE XREF: string_compare+16j
mov ax, 1
jmp short loc_2B
; ---------------------------------------------------------------
-------------
loc_27: ; CODE XREF: string_compare+18j
inc bx
inc si
jmp short loc_12
; ---------------------------------------------------------------
-------------
loc_2B: ; CODE XREF: string_compare+12j
; string_compare+1Dj
pop si
pop bp
retn 4
string_compare endp
string_compare_far proc near ; CODE XREF: WinMain+18p
arg_0 = word ptr 4
arg_2 = word ptr 6
arg_4 = word ptr 8
arg_6 = word ptr 0Ah
push bp
mov bp, sp
push si
mov si, [bp+arg_0]
mov bx, [bp+arg_4]
loc_3A: ; CODE XREF: string_compare_far+35j
mov es, [bp+arg_6]
mov al, es:[bx]
mov es, [bp+arg_2]
cmp al, es:[si]
jz short loc_4C
xor ax, ax
jmp short loc_67
; ---------------------------------------------------------------
```

```
-------------
loc_4C: ; CODE XREF: string_compare_far+16j
mov es, [bp+arg_6]
cmp byte ptr es:[bx], 0
jz short loc_5E
mov es, [bp+arg_2]
cmp byte ptr es:[si], 0
jnz short loc_63
loc_5E: ; CODE XREF: string_compare_far+23j
mov ax, 1
jmp short loc_67
; ---------------------------------------------------------------
-------------
loc_63: ; CODE XREF: string_compare_far+2Cj
inc bx
inc si
jmp short loc_3A
; ---------------------------------------------------------------
-------------
loc_67: ; CODE XREF: string_compare_far+1Aj
; string_compare_far+31j
pop si
pop bp
retn 8
string_compare_far endp
remove_digits proc near ; CODE XREF: WinMain+1Fp
arg_0 = word ptr 4
push bp
mov bp, sp
mov bx, [bp+arg_0]
loc_72: ; CODE XREF: remove_digits+18j
mov al, [bx]
test al, al
jz short loc_86
cmp al, 30h ; '0'
jb short loc_83
cmp al, 39h ; '9'
ja short loc_83
mov byte ptr [bx], 2Dh ; '-'
loc_83: ; CODE XREF: remove_digits+Ej
; remove_digits+12j
inc bx
jmp short loc_72
; ---------------------------------------------------------------
-------------
loc_86: ; CODE XREF: remove_digits+Aj
pop bp
retn 2
remove_digits endp
WinMain proc near ; CODE XREF: start+EDp
push bp
mov bp, sp
mov ax, offset aAsd ; "asd"
```

```
push ax
mov ax, offset aDef ; "def"
push ax
call string_compare
push ds
mov ax, offset aAsd ; "asd"
push ax
push ds
mov ax, offset aDef ; "def"
push ax
call string_compare_far
mov ax, offset aHello1234World ; "hello 1234 world"
push ax
call remove_digits
xor ax, ax
push ax
push ds
mov ax, offset aHello1234World ; "hello 1234 world"
push ax
push ds
mov ax, offset aCaption ; "caption"
push ax
mov ax, 3 ; MB_YESNOCANCEL
push ax
call MESSAGEBOX
xor ax, ax
pop bp
retn 0Ah
WinMain endp
```

我们可以看到所谓的“near”指针和“far”指针：另一个奇怪的16位8086现象。 可以在70章继续读到相关内容。 近指针就是那些指向当前数据段内的指针。因为，string_compare()函数仅仅用到2个16位指针，而且访问数据通过DS指向了它（mov al, es:[bx]）。远指针也同样在我的16位MessageBox()例子里面：见30.2节。 因此，在访问文本时，Windows内核并不关心使用那个数据段，所以它需要更完整的信息。 使用这种区别的原因可能是因为紧凑的程序可能使用仅仅一个64kb的数据段。所以他并不需要传递地址的高位数据，因为它们永远是不变的。大一点的程序可能会使用多个64kb数据段，所以它们每次操作都需要需要区分它们是在哪个数据段里面。 对代码段来说也是相同的故事，比较短小的程序可能在64k的数据段里面包含有所有的可执行代码，然后所有的函数都会由CALL NEAR来调用，代码使用RETN返回。但是，如果有多个代码段的话，函数地址就会按对区分，然后使用CALL FAR来调用，代码会使用RETF返回。 这就是在编译器中指定“内存模型”会发生的事情。 MS-DOS和Win16编译器针对每个内存模型都有有特别的库：它们会因为数据和代码的不同的指针模型而不同。

53.6 例子#6

```
#include <windows.h>
#include <time.h>
#include <stdio.h>
char strbuf[256];

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    struct tm *t;
    time_t unix_time;
    unix_time=time(NULL);
    t=localtime (&unix_time);
    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year
+1900, t->tm_mon, t->tm_mday,
    t->tm_hour, t->tm_min, t->tm_sec);
    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};
```

```
WinMain proc near
var_4 = word ptr -4
var_2 = word ptr -2
push bp
mov bp, sp
push ax
push ax
xor ax, ax
call time_
mov [bp+var_4], ax ; low part of UNIX time
mov [bp+var_2], dx ; high part of UNIX time
lea ax, [bp+var_4] ; take a pointer of high part
call localtime_
mov bx, ax ; t
push word ptr [bx] ; second
push word ptr [bx+2] ; minute
push word ptr [bx+4] ; hour
push word ptr [bx+6] ; day
push word ptr [bx+8] ; month
mov ax, [bx+0Ah] ; year
add ax, 1900
push ax
mov ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
push ax
mov ax, offset strbuf
push ax
call sprintf_
add sp, 10h
xor ax, ax ; NULL
push ax
push ds
mov ax, offset strbuf
push ax
push ds
mov ax, offset aCaption ; "caption"
push ax
xor ax, ax ; MB_OK
push ax
call MESSAGEBOX
xor ax, ax
mov sp, bp
pop bp
retn 0Ah
WinMain endp
```

UNIX时间是32位的，所以它返回在DX:AX寄存器对中，而且将他们存储到两个本地16位变量中。然后一个指向值对的指针会被当作参数传给localtime()函数。Localtime()函数有一个struct tm，它将通过C库分配内存，所以只有指向它的指针返回了。顺便一提，这也意味着在它的结果被使用之前，函数不能被再次调用。 对time()和localtime()两个函数来说，Watcom调用转换将会在这里：前四个参数使用

AX、DX、BX、CX传递，剩余的通过栈来传递。使用这个转换的函数也会在名字最后使用下划线来标记。 Sprintf()并不使用PASCAL调用转换，也不会使用watcom转换，所以参数将使用寻常的cdecl方式传递（47.1节）。

53.6.1 全局变量

这里用同样的例子，但是变量是全局变量：

```
#include <windows.h>
#include <time.h>
#include <stdio.h>
char strbuf[256];
struct tm *t;
time_t unix_time;

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    unix_time=time(NULL);
    t=localtime (&unix_time);
    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year
+1900, t->tm_mon, t->tm_mday,
    t->tm_hour, t->tm_min, t->tm_sec);
    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};
```

```
unix_time_low dw 0
unix_time_high dw 0
t dw 0
WinMain proc near
push bp
mov bp, sp
xor ax, ax
call time_
mov unix_time_low, ax
mov unix_time_high, dx
mov ax, offset unix_time_low
call localtime_
mov bx, ax
mov t, ax ; will not be used in future...
push word ptr [bx] ; seconds
push word ptr [bx+2] ; minutes
push word ptr [bx+4] ; hour
push word ptr [bx+6] ; day
push word ptr [bx+8] ; month
mov ax, [bx+0Ah] ; year
add ax, 1900
push ax
mov ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
push ax
mov ax, offset strbuf
push ax
call sprintf_
add sp, 10h
xor ax, ax ; NULL
push ax
push ds
mov ax, offset strbuf
push ax
push ds
mov ax, offset aCaption ; "caption"
push ax
xor ax, ax ; MB_OK
push ax
call MESSAGEBOX
xor ax, ax ; return 0
pop bp
retn 0Ah
WinMain endp
```

T不会被使用，但是编译器还是用代码存储了这个值。因为他并不确定，也许这个值会在某个地方被用到。

Part IV JAVA

第五十四章

JAVA

54.1介绍

大家都知道，java有很多的反编译器（或是产生JVM字节码） 原因是JVM字节码比其他的X86低级代码更容易进行反编译。

- 多很多相关数据类型的信息。
- JVM（java虚拟机）内存模型更严格和概括。
- java编译器没有做任何的优化工作（JVM JIT不是实时），所以，类文件中的字节代码的通常更清晰易读。

JVM字节码知识什么时候有用呢？

- 文件的快速粗糙的打补丁任务，类文件不需要重新编译反编译的结果。
- 分析混淆代码
- 创建你自己的混淆器。
- 创建编译器代码生成器（后端）目标。

我们从一段简短的代码开始，除非特殊声明，我们用的都是JDK1.7

反编译类文件使用的命令，随处可见：javap -c -verbase.

在这本书中提供的很多的例子，都用到了这个。

54.2 返回一个值

可能最简单的java函数就是返回一些值，oh，并且我们必须注意，一边情况下，在java中没有孤立存在的函数，他们是“方法”(method)，每个方法都是被关联到某些类，所以方法不会被定义在类外面， 但是我还是叫他们“函数” (function),我这么用。

```
public class ret
{
public static int main(String[] args)
{
return 0;
}
}
```

编译它。

```
javac ret.java
```

。。。使用Java标准工具反编译。

```
javap -c -verbose ret.class
```

会得到结果：

```
public static int main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iconst_0
1: ireturn
```

对于java开发者在编程中，0是使用频率最高的常量。 因为区分短一个短字节的iconst_0指令入栈0，iconst_1指令（入栈），iconst_2等等，直到iconst5。也可以有iconst_m1, 推送-1。

就像在MIPS中，分离一个寄存器给0常数：3.5.2 在第三页。

栈在JVM中用于在函数调用时，传参和传返回值。因此， iconst_0是将0入栈，ireturn指令，（i就是integer的意思。）是从栈顶返回整数值。

〔校准到这,未完待续...〕

让我们写一个简单的例子， 现在我们返回1234：

```
public class ret
{
public static int main(String[] args)
{
return 1234;
}
}
```

我们得到：

清单： 54.2:jdk1.7(节选) public static int main(java.lang.String[]); flags: ACC_PUBLIC, ACC_STATIC Code: stack=1, locals=1, args_size=1 0: sipush 1234 3: ireturn

sipush(shot integer)如栈值是1234,slot的名字以为着一个16bytes值将会入栈。sipush(短整型) 1234数值确认时候16-bit值。

```
public class ret
{
public static int main(String[] args)
{
return 12345678;
}
}
```

更大的值是什么？

清单 54.3 常量区

```
...
#2 = Integer 12345678
...
```

5栈顶

```
public static int main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATI
Code:
stack=1, locals=1, args_size=1
0: ldc #2 // int 12345678
2: ireturn
```

操作码 JVM的指令码操作码不可能编码成32位数，开发者放弃这种可能。因此，32位数字12345678是被存储在一个叫做常量区的地方。让我们说（大多数被使用的常数（包括字符，对象等等车）） 对我们而言。

对JVM来说传递常量不是唯一的，MIPS ARM和其他的RISC CPUS也不可能把32位操作编码成32位数字，因此 RISC CPU（包括MIPS和ARM）去构造一个值需要一系列的步骤，或是他们保存在数据段中： 28。3 在654页.291 在695页。

MIPS码也有一个传统的常量区，literal pool(原语区) 这个段被叫做"lit4"(对于32位单精度浮点数常数存储) 和lit8(64位双精度浮点整数常量区)

布尔型

```
public class ret
{
public static boolean main(String[] args)
{
return true;
}
}



public static boolean main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iconst_1
```

这个JVM字节码是不同于返回的整数学 ，32位数据，在形参中被当成逻辑值使用。像C/C++，但是不能像使用整型或是viceversa返回布尔型，类型信息被存储在类文件中，在运行时检查。

16位短整型也是一样。

```
public class ret
{

public static short main(String[] args)
{
return 1234;
}
}
public static short main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: sipush 1234
3: ireturn
```

还有char 字符型？

```
public class ret
{
public static char main(String[] args)
{
return 'A';
}
}
public static char main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: bipush 65
2: ireturn
```

bipush 的意思"push byte"字节入栈，不必说java的char是16位UTF16字符，和short 短整型相等，单ASCII码的A字符是65，它可能使用指令传输字节到栈。

让我们是试一下byte。

```
public class retc
{
public static byte main(String[] args)
{
return 123;
}
}
public static byte main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC

Code:
stack=1, locals=1, args_size=1
0: bipush 123
2: ireturn
```

也许会问，位什么费事用两个16位整型当32位用？为什么char数据类型和短整型类型还使用char.

答案很简单，为了数据类型的控制和代码的可读性。char也许本质上short相同，但是我们快速的掌握它的占位符，16位的UTF字符，并且不像其他的integer值符。使用 short,为各位展现一下变量的范围被限制在16位。在需要的地方使用boolean型也是一个很好的主意。代替C样式的int也是为了相同的目的。

在java中integer的64位数据类型。

```
public class ret3
{
public static long main(String[] args)
{
return 1234567890123456789L;
}
}
```

清单54.4常量区

```
...
#2 = Long 1234567890123456789l
...
public static long main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: ldc2_w #2 // long ⤦
Ç 1234567890123456789l
3: lreturn
```

64位数也被在存储在常量区，ldc2_w 加载它，lreturn返回它。 ldc2_w指令也是从内存常量区中加载双精度浮点数。（同样占64位）

```
public class ret
{
public static double main(String[] args)
{
return 123.456d;
}
}
```

清单54.5常量区

```
...
#2 = Double 123.456d
...
public static double main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: ldc2_w #2 // double 123.456⤦
Ç d
3: dreturn
```

dreturn 代表 "return double"

最后，单精度浮点数：

```
public class ret
{
public static float main(String[] args)
{
return 123.456f;
}
}
```

清单54.6 常量区

```
...
#2 = Float 123.456f
...
public static float main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: ldc #2 // float 123.456f
2: freturn
```

此处的ldc指令使用和32位整型数据一样，从常量区中加载。freturn 的意思是"return float"

那么函数还能返回什么呢？

```
public class ret
{
public static void main(String[] args)
{
return;
}
}
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=0, locals=1, args_size=1
0: return
```

这以为着，使用return控制指令确没有返回实际的值，知道这一点就非常容易的从最后一条指令中演绎出函数（或是方法）的返回类型。

54.3 简单的计算函数

让我们继续看简单的计算函数。

```
public class calc
{
public static int half(int a)
{
return a/2;
}
}
```

这种情况使用icont_2会被使用。

```
public static int half(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: iload_0
1: iconst_2
2: idiv
3: ireturn
```

iload_0 将零给函数做参数，然后将其入栈。iconst_2将2入栈，这两个指令执行后，栈看上去是这个样子的。

```
+---+
TOS ->| 2 |
+---+
| a |
+---+
```

idiv携带两个值在栈顶， divides 只有一个值，返回结果在栈顶。

```
+--------+
TOS ->| result |
+--------+
```

ireturn取得比返回。 让我们处理双精度浮点整数。

```
public class calc
{
public static double half_double(double a)
{
return a/2.0;
}
}
```

清单54.7 常量区

```
...
#2 = Double 2.0d
...
public static double half_double(double);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=2, args_size=1
0: dload_0
1: ldc2_w #2 // double 2.0d
4: ddiv
5: dreturn
```

类似，只是ldc2_w指令是从常量区装载2.0，另外，所有其他三条指令有d前缀，意思是他们工作在double数据类型下。

我们现在使用两个参数的函数。

```
public class calc
{
public static int sum(int a, int b)
{
return a+b;
}
}
public static int sum(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: iload_0
1: iload_1
2: iadd
3: ireturn
```

iload_0加载第一个函数参数（a)，iload_2 第二个参数(b)下面两条指令执行后，栈的情况如下：

```
+---+
TOS ->| b |
+---+
| a |
+---+
```

iadds 增加两个值，返回结果在栈顶。 +--------+ TOS ->| result | +--------+

让我们把这个例子扩展成长整型数据类型。

```
public static long lsum(long a, long b)
{
return a+b;
}
```

我们得到的是：

```
public static long lsum(long, long);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=4, args_size=2
0: lload_0
1: lload_2
2: ladd
3: lreturn
```

第二个（load指令从第二参数槽中，取得第二参数。这是因为64位长整型的值占用来位，用了另外的话2位参数槽。）

稍微复杂的例子

```
public class calc
{
public static int mult_add(int a, int b, int c)
{
return a*b+c;
}
}
public static int mult_add(int, int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=3, args_size=3
0: iload_0
1: iload_1
2: imul
3: iload_2
4: iadd
5: ireturn
```

第一是相乘，积被存储在栈顶。

```
+---------+
TOS ->| product |
+---------+
```

iload_2加载第三个参数（C）入栈。

```
+---------+
TOS ->| c |
+---------+
| product |
+---------+
```

现在iadd指令可以相加两个值。

54.4 JVM内存模型

X86和其他低级环境系统使用栈传递参数和存储本地变量，JVM稍微有些不同。

主要体现在： 本地变量数组（LVA）被用于存储到来函数的参数和本地变量。iload_0指令是从其中加载值，istore存储值在其中，首先，函数参数到达：开始从0或者1(如果0参被this指针用。)，那么本地局部变量被分配。

每个槽子的大小都是32位，因此long和double数据类型都占两个槽。

操作数栈（或只是"栈"），被用于在其他函数调用时，计算和传递参数。不像低级X86的环境，它不能去访问栈，而又不明确的使用pushes和pops指令，进行出入栈操作。

54.5 简单的函数调用

mathrandom()返回一个伪随机数，函数范围在「0.0...1.0)之间，但对我们来说，由于一些原因，我们常常需要设计一个函数返回数值范围在「0.0...0.5)

```
public class HalfRandom
{
public static double f()
{
return Math.random()/2;
}
}
```

54.8 常量区

```
...
#2 = Methodref #18.#19 // java/lang/Math.⤦
Ç random:()D
6(Java) Local Variable Array

#3 = Double 2.0d
...
#12 = Utf8 ()D
...
#18 = Class #22 // java/lang/Math
#19 = NameAndType #23:#12 // random:()D
#22 = Utf8 java/lang/Math
#23 = Utf8 random
public static double f();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=0, args_size=0
0: invokestatic #2 // Method java/⤦
Ç lang/Math.random:()D
3: ldc2_w #3 // double 2.0d
6: ddiv
7: dreturn
```

java本地变量数组 916 静态执行调用math.random()函数，返回值在栈顶。结果是被0.5初返回的，但函数名是怎么被编码的呢？ 在常量区使用methodres表达式,进行编码的，它定义类和方法的名称。第一个methodref 字段指向表达式，其次，指向通常文本字符（"java/lang/math"） 第二个methodref表达指向名字和类型表达式，同时链接两个字符。第一个方法的名字式字符串"random",第二个字符串是"()D",来编码函数类型，它以为这两个值（因此D是字符串）这种方式1JVM可以检查数据类型的正确性：2）java反编译器可以从被编译的类文件中修改数据类型。

最后，我们试着使用"hello，world！"作为例子。

```
public class HelloWorld
{
public static void main(String[] args)
{
System.out.println("Hello, World");
}
}
```

54.9 常量区

常量区的ldc行偏移3，指向"hello，world！"字符串，并且将其入栈，在java里它被成为饮用，其实它就是指针，或是地址。

```
...
#2 = Fieldref #16.#17 // java/lang/System.⤦
Ç out:Ljava/io/PrintStream;
#3 = String #18 // Hello, World
#4 = Methodref #19.#20 // java/io/⤦
Ç PrintStream.println:(Ljava/lang/String;)V
...
#16 = Class #23 // java/lang/System
#17 = NameAndType #24:#25 // out:Ljava/io/⤦
Ç PrintStream;
#18 = Utf8 Hello, World
#19 = Class #26 // java/io/⤦
Ç PrintStream
#20 = NameAndType #27:#28 // println:(Ljava/⤦
Ç lang/String;)V
...
#23 = Utf8 java/lang/System
#24 = Utf8 out
#25 = Utf8 Ljava/io/PrintStream;
#26 = Utf8 java/io/PrintStream
#27 = Utf8 println
#28 = Utf8 (Ljava/lang/String;)V
...
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
3: ldc #3 // String Hello, ⤦
Ç World
5: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
8: return
```

常见的invokevirtual指令，从常量区取信息，然后调用pringln()方法，貌似我们知道的println()方法，适用于各种数据类型，我这种println()函数版本，预先给的是字符串类型。

但是第一个getstatic指令是干什么的？这条指令取得对象信息的字段的一个引用或是地址。输出并将其进栈，这个值实际更像是println放的指针，因此，内部的print method取得两个参数，输入1指向对象的this指针，2）"hello，world"字符串的地址，确实，println()在被初始化系统的调用，对象之外，为了方便，javap使用工具把所有的信息都写入到注释中。

54.6 调用beep()函数

这可能是最简单的，不使用参数的调用两个函数。

```
public static void main(String[] args)
{
java.awt.Toolkit.getDefaultToolkit().beep();
};
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: invokestatic #2 // Method java/↩
Ç awt/Toolkit.getDefaultToolkit:()Ljava/awt/Toolkit;
3: invokevirtual #3 // Method java/↩
Ç awt/Toolkit.beep:()V
6: return
```

首先，invokestatic在0行偏移调用javaawt.toolkit. getDefaultTookKit()函数,返回toolkit类对象的引用，invokedvirtualIFge指令在3行偏移，调用这个类的beep（）方法。

54.7 线性同余伪随机数生成器

我们来试一个简单的伪随机函数生成器，我已经在这本书中用过一次了。（在500页20行）

```
public class LCG
{
public static int rand_state;
public void my_srand (int init)
{
rand_state=init;
}
public static int RNG_a=1664525;
public static int RNG_c=1013904223;

public int my_rand ()
{
rand_state=rand_state*RNG_a;
rand_state=rand_state+RNG_c;
return rand_state & 0x7fff;
}
}
```

一对类的字段，在最开始时被初始化。但是怎么能，在javap的输出中，发现类的构造呢？

```
static {};
flags: ACC_STATIC
Code:
stack=1, locals=0, args_size=0
0: ldc #5 // int 1664525
2: putstatic #3 // Field RNG_a:I
5: ldc #6 // int 1013904223
7: putstatic #4 // Field RNG_c:I
10: return
```

这种变量的初始化，RNG_a占用了3个参数槽，iRNG_C是4个，而puststatic指令是，用于设定常量。

my_srand()函数，只是将输入值，存储到rand_state中;

```
public void my_srand(int);
flags: ACC_PUBLIC
Code:
stack=1, locals=2, args_size=2
0: iload_1
1: putstatic #2 // Field ⤦
Ç rand_state:I
4: return
```

iload_1 取得输入值并将其入栈。但为什么不用iload_0?因为这个函数可能使用类的字段属性，因此这个变量被作为参数0传递给了函数，rand_state字段属性，在类中占用2个参数槽子。

现在my_rand():

```
public int my_rand();
flags: ACC_PUBLIC
Code:
stack=2, locals=1, args_size=1
0: getstatic #2 // Field ⤦
Ç rand_state:I
3: getstatic #3 // Field RNG_a:I
6: imul
7: putstatic #2 // Field ⤦
Ç rand_state:I
10: getstatic #2 // Field ⤦
Ç rand_state:I
13: getstatic #4 // Field RNG_c:I
16: iadd
17: putstatic #2 // Field ⤦
Ç rand_state:I
20: getstatic #2 // Field ⤦
Ç rand_state:I
23: sipush 32767
26: iand
27: ireturn
```

它仅是加载了所有对象字段的值。在20行偏移，操作和更新rand_state，使用putstatic指令。

rand_state 值被再次重载（因为之前，使用过putstatic指令，其被从栈中弃出）这种代码其实比较低效率，但是可以肯定的是，JVM会经常的，对其进行很好的优化。

54.8 条件跳转

让我们进入条件跳转

```
public class abs
{
public static int abs(int a)
{
if (a<0)
return -a;
return a;
}
}
public static int abs(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iload_0
1: ifge 7
4: iload_0
5: ineg
6: ireturn
7: iload_0
8: ireturn
```

ifge跳转到7行偏移，如果栈顶的值大于等于0，别忘了，任何IFXX指令从栈中pop出栈值（用于进行比较）

另外一个例子

```
public static int min (int a, int b)
{
if (a>b)
return b;
return a;
}
```

我们得到的是：

```
public static int min(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: iload_0
1: iload_1
2: if_icmple 7
5: iload_1
6: ireturn
7: iload_0
8: ireturn
```

if_icmple出栈两个值并比较他们，如果第三个子值比第一个值小（或者等于）发生跳转到行偏移7.

当我们定义max()函数。

```
public static int max (int a, int b)
{
if (a>b)
return a;
return b;
}
```

。。。结果代码是是一样的，但是最后两个iload指令（行偏移5和行偏移7）被跳转了。

```
public static int max(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: iload_0
1: iload_1
2: if_icmple 7
5: iload_0
6: ireturn
7: iload_1
8: ireturn
```

更复杂的例子。。

```
public class cond
{
public static void f(int i)
{
if (i<100)
System.out.print("<100");
if (i==100)
System.out.print("==100");
if (i>100)
System.out.print(">100");
if (i==0)
System.out.print("==0");
}
}
public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: iload_0
1: bipush 100
3: if_icmpge 14
6: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
9: ldc #3 // String <100
11: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.print:(Ljava/lang/String;)V
14: iload_0
15: bipush 100
17: if_icmpne 28
20: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
23: ldc #5 // String ==100
25: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.print:(Ljava/lang/String;)V
28: iload_0
29: bipush 100
31: if_icmple 42
34: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
37: ldc #6 // String >100
39: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.print:(Ljava/lang/String;)V
42: iload_0
43: ifne 54
46: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
49: ldc #7 // String ==0
51: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.print:(Ljava/lang/String;)V
54: return
```

if_icmpge出栈两个值，并且比较它们，如果第的二个值大于第一个，发生跳转到行偏移14，if_icmpne和if_icmple做的工作类似，但是使用不同的判断条件。

在行偏移43的ifne指令，它的名字不是很恰当，我要愿意把它命名为ifnz

如果栈定的值不是0跳转，但是这是怎么做的，总跳转到行偏移54，如果输入的值不是另，如果是0，执行流程进入行偏移46，“==”字符串被打印。

N.BJVM没有无符号数据类型，所以，比较指令的操作数，只有还有符号整数值。

54.9 传递参数值

我们来扩展一下min()/max()这个例子。

```
public class minmax
{
public static int min (int a, int b)
{
if (a>b)
return b;
return a;
}
public static int max (int a, int b)
{
if (a>b)
return a;
return b;
}
public static void main(String[] args)
{
int a=123, b=456;
int max_value=max(a, b);
int min_value=min(a, b);
System.out.println(min_value);
System.out.println(max_value);
}
}
```

这是main()函数的代码。

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=5, args_size=1
0: bipush 123
2: istore_1
3: sipush 456
6: istore_2
7: iload_1
8: iload_2
9: invokestatic #2 // Method max:(II↙
Ç )I
12: istore_3
13: iload_1
14: iload_2
15: invokestatic #3 // Method min:(II↙
Ç )I
18: istore 4
20: getstatic #4 // Field java/↙
Ç lang/System.out:Ljava/io/PrintStream;
23: iload 4
25: invokevirtual #5 // Method java/io↙
Ç /PrintStream.println:(I)V
28: getstatic #4 // Field java/↙
Ç lang/System.out:Ljava/io/PrintStream;
31: iload_3
32: invokevirtual #5 // Method java/io↙
Ç /PrintStream.println:(I)V
35: return
```

参数在栈中的被传给其他函数，返回值在栈顶。

54.10位。

所有位操作工作，与其他的一些ISA（指令集架构）类似：

```
public static int set (int a, int b)
{
return a | 1<<b;
}
public static int clear (int a, int b)
{
return a & (~(1<<b));
}
public static int set(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=2
0: iload_0
1: iconst_1
2: iload_1
3: ishl
4: ior
5: ireturn
public static int clear(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=2
0: iload_0
1: iconst_1
2: iload_1
3: ishl
4: iconst_m1
5: ixor
6: iand
7: ireturn
```

iconst_m1将-1入栈，这数其实就是16进制的0xFFFFFFFF，将0xFFFFFFFF作为XOR-ing指令执行的操作数。起到的效果就是把所有bits位反向，（A.6.2在1406页）

我将所有数据类型，扩展成64为长整型。

```
public static long lset (long a, int b)
{
return a | 1<<b;
}
public static long lclear (long a, int b)
{
return a & (~(1<<b));
}
public static long lset(long, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=3, args_size=2
0: lload_0
1: iconst_1
2: iload_2
3: ishl
4: i2l
5: lor
6: lreturn
public static long lclear(long, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=3, args_size=2
0: lload_0
1: iconst_1
2: iload_2
3: ishl
4: iconst_m1
5: ixor
6: i2l
7: land
8: lreturn
```

代码是相同的，但是指令前面使用了前缀L，操作64位值，并且第二个函数参数还是int类型，并且32值需要升级为64位值，值被i21指令使用，本质上 就是把整型，扩展成64位长整型.

54.11循环

```
public class Loop
{
public static void main(String[] args)
{
for (int i = 1; i <= 10; i++)
{
System.out.println(i);
}
}
}
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=1
0: iconst_1
1: istore_1
2: iload_1
3: bipush 10
5: if_icmpgt 21
8: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
11: iload_1
12: invokevirtual #3 // Method java/io⤦
Ç /PrintStream.println:(I)V
15: iinc 1, 1
18: goto 2
21: return
```

icont_1将1推入到栈顶，istore_1将其存入到LVA的参数槽1，为什么没有零槽？因为main()函数只有一个参数，并且指向其的引用，就在第0号槽中。

因此，i本地变量总是在1号参数槽中。 指令在行3偏移和行5偏移，将i和10的比较。如果i大，执行流进入行21偏移，函数结束了，如果不被println调用。i在行11偏移进行了重新加载，之后给println使用。

多说一句，我们调用pringln打印数据类型是整型，我们看注释，“i，v”，i的意思是整型，v的意思是返回void。

当println函数结束，i是步进到行15偏移，指令第一个操作数是参数槽1的值。第二个是数值1与本地变量相加结果。

goto指令就是跳转，它跳转到循环体的开始地址，再行偏移2.

让我们进行更复杂的例子。

```
public class Fibonacci
{
public static void main(String[] args)
{
int limit = 20, f = 0, g = 1;
for (int i = 1; i <= limit; i++)
{
f = f + g;
g = f - g;
System.out.println(f);
}
}
}
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=5, args_size=1
0: bipush 20
2: istore_1
3: iconst_0
4: istore_2
5: iconst_1
6: istore_3
7: iconst_1
8: istore 4
10: iload 4
12: iload_1
13: if_icmpgt 37
16: iload_2
17: iload_3
18: iadd
19: istore_2
20: iload_2
21: iload_3
22: isub
23: istore_3
24: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
27: iload_2
28: invokevirtual #3 // Method java/io⤦
Ç /PrintStream.println:(I)V
31: iinc 4, 1
34: goto 10
37: return
```

LVA槽中参数映射。 0-main（）的唯一参数。 1-限制，总是20. 2-f 3-g 4-i

我们可以看到java编译器在LVA参数槽分配变量，并且是相同的顺序，就像在源代码中声明变量。

分离指令istore，是用于访问参数槽0123，但是不能大于4，因此，附加一些操作，在行2，8偏移，使用槽中数据作为操作数，类似于在偏移10位置的iload指令。

无可口非，分离其他的槽，限制变量总是20（其本质上就是一个常数），重加载值很经常吗？

JVM JIT 编译器经常可以对其优化的很好。在代码中人工的干预优化其实是没有什么太大价值的。

54.12 switch()函数

switch（）语句的实现是用tableswitch指令， public static void f(int a) { switch (a) { case 0: System.out.println("zero"); break; case 1: System.out.println("one\n"); break; case 2: System.out.println("two\n"); break; case 3: System.out.println("three\n"); break; case 4: System.out.println("four\n"); break; default: System.out.println("something unknown\⤦ Ç n"); break; }; }

尽可能简单的例子

```
public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: iload_0
1: tableswitch { // 0 to 4
0: 36
1: 47
2: 58
3: 69
4: 80
default: 91
}
36: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
39: ldc #3 // String zero
41: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
44: goto 99
47: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
50: ldc #5 // String one\n
52: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
55: goto 99
58: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
61: ldc #6 // String two\n
63: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
66: goto 99
```

```
69: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
72: ldc #7 // String three\n
74: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
77: goto 99
80: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
83: ldc #8 // String four\n
85: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
88: goto 99
91: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
94: ldc #9 // String ⤦
Ç something unknown\n
931
CHAPTER 54. JAVA 54.13. ARRAYS
96: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
99: return
```

54.13数组

54.13.1简单的例子

我们首先创建一个长度是10的整型的数组，对其初始化。

```
public static void main(String[] args)
{
int a[]=new int[10];
for (int i=0; i<10; i++)
a[i]=i;
dump (a);
}
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=3, args_size=1
0: bipush 10
2: newarray int
4: astore_1
5: iconst_0
6: istore_2
7: iload_2
8: bipush 10
10: if_icmpge 23
13: aload_1
14: iload_2
15: iload_2
16: iastore
17: iinc 2, 1
20: goto 7
23: aload_1
24: invokestatic #4 // Method dump:([
Ç I)V
27: return
```

newarray指令，创建了一个有10个整数元素的数组，数组的大小设置使用bipush指令，然后结果会返回到栈顶。数组类型用newarry指令操作符，进行设定。

newarray被执行后，引用（指针）到新创建的数据，栈顶的槽中，astore_1存储引用指向到LVA的一号槽，main()函数的第二个部分，是循环的存储值1到相应的素组元素。 aload_1得到数据的引用并放入到栈中。lastore将integer值从堆中存储到素组中，引用当前的栈顶。main()函数代用dump()的函数部分，参数是，准备给aload_1指令的（行偏移23）

现在我们进入dump()函数。

```
public static void dump(int a[])
{
for (int i=0; i<a.length; i++)
System.out.println(a[i]);
}
public static void dump(int[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1
0: iconst_0
1: istore_1
2: iload_1
3: aload_0
4: arraylength
5: if_icmpge 23
8: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
11: aload_0
12: iload_1
13: iaload
14: invokevirtual #3 // Method java/io⤦
Ç /PrintStream.println:(I)V
17: iinc 1, 1
20: goto 2
23: return
```

到了引用的数组在0槽，a.length表达式在源代码中是转化到arraylength指令，它取得数组的引用，并且数组的大小在栈顶。 iaload在行偏移13被用于装载数据元素。它需要在堆栈中的数组引用。用aload_0 11并且索引（用iload_1在行偏移12准备）

无可厚非，指令前缀可能会被错误的理解，就像数组指令，那样不正确，这些指令和对象的引用一起工作的。数组和字符串都是对象。

54.13.2 数组元素的求和

另外的例子

```
public class ArraySum
{
public static int f (int[] a)
{
int sum=0;
for (int i=0; i<a.length; i++)
sum=sum+a[i];
return sum;
}
}
public static int f(int[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=3, args_size=1
0: iconst_0
1: istore_1
2: iconst_0
3: istore_2
4: iload_2
5: aload_0
6: arraylength
7: if_icmpge 22
10: iload_1
11: aload_0
12: iload_2
13: iaload
14: iadd
15: istore_1
16: iinc 2, 1
19: goto 4
22: iload_1
23: ireturn
```

LVA槽0是数组的引用，LVA槽1是本地变量和。

54.13.3 main（）函数唯一的数据参数

让我们使用唯一的main()函数参数，字符串数组。

```
public class UseArgument
{
public static void main(String[] args)
{
System.out.print("Hi, ");
System.out.print(args[1]);
System.out.println(". How are you?");
}
}
```

0参（argument）第0个参数是程序（和C/C++类似）

因此第一个参数，而第一参数是拥护提供的。

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=1, args_size=1
0: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
3: ldc #3 // String Hi,
5: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.print:(Ljava/lang/String;)V
8: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
11: aload_0
12: iconst_1
13: aaload
14: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.print:(Ljava/lang/String;)V
17: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
20: ldc #5 // String . How ⤦
Ç are you?
22: invokevirtual #6 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
25: return
```

aload_0在11行加载，第0个LVA槽的引用（main（）函数唯一的参数）iconst_1和aload在行偏移12,13，取得数组第一个元素的引用（从0计数） 字符串对象的引用在栈顶行14行偏移，给println方法。

54.1.34 初始化字符串数组

```
class Month
{

public static String[] months =
{
"January",
"February",
"March",
"April",
"May",
"June",
"July",
"August",
"September",
"October",
"November",
"December"
};
public String get_month (int i)
{
return months[i];
};
}
```

get_month()函数很简单

```
public java.lang.String get_month(int);
flags: ACC_PUBLIC
Code:
stack=2, locals=2, args_size=2
0: getstatic #2 // Field months:[⤦
Ç Ljava/lang/String;
3: iload_1
4: aaload
5: areturn
```

aaload操作数组引用，java字符串是一个对象，所以a_instructiong被用于操作他们.areturn返回字符串对象的引用。

month[]数值是如果初始化的？

```
static {};
flags: ACC_STATIC
Code:
stack=4, locals=0, args_size=0
0: bipush 12
2: anewarray #3 // class java/⤦
Ç lang/String
5: dup
```

```
6: iconst_0
7: ldc #4 // String January
9: aastore
10: dup
11: iconst_1
12: ldc #5 // String ⤦
Ç February
14: aastore
15: dup
16: iconst_2
17: ldc #6 // String March
19: aastore
20: dup
21: iconst_3
22: ldc #7 // String April
24: aastore
25: dup
26: iconst_4
27: ldc #8 // String May
29: aastore
30: dup
31: iconst_5
32: ldc #9 // String June
34: aastore
35: dup
36: bipush 6
38: ldc #10 // String July
40: aastore
41: dup
42: bipush 7
44: ldc #11 // String August
46: aastore
47: dup
48: bipush 8
50: ldc #12 // String ⤦
Ç September
52: aastore
53: dup
54: bipush 9
56: ldc #13 // String October
58: aastore
59: dup
60: bipush 10
62: ldc #14 // String ⤦
Ç November
64: aastore
65: dup
66: bipush 11
68: ldc #15 // String ⤦
Ç December
70: aastore
71: putstatic #2 // Field months:[⤦
Ç Ljava/lang/String;
```

```
74: return
```

anewarray 创建一个新数组的引用（a是一个前缀）对象的类型被定义在anewarray操作数中，它在这是“java/lang/string”文本字符串，在这之前的bipush 1L是设置数组的大小。 对于我们再这看到一个新指令dup，他是一个众所周知的堆栈操作的计算机指令。用于复制栈顶的值。（包括了之后的编程语言）它在这是用于复制数组的引用。因为aastore张玲玲 起到弹出堆栈中的数组的作用，但是之后，aastore需要在使用一次，java编译器，最好同dup代替getstatic指令，用于生成之前的每个数组的存贮操作。例如，月份字段。

54.13.5可变参数

可变参数 变长参数函数，实际上使用的就是数组，实际使用的就是数组。

```
public static void f(int... values)
{
for (int i=0; i<values.length; i++)
System.out.println(values[i]);
}
public static void main(String[] args)
{
f (1,2,3,4,5);
}
public static void f(int...);
flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
Code:
stack=3, locals=2, args_size=1
0: iconst_0
1: istore_1
2: iload_1
3: aload_0
4: arraylength
5: if_icmpge 23
8: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
11: aload_0
12: iload_1
13: iaload
14: invokevirtual #3 // Method java/io⤦
Ç /PrintStream.println:(I)V
17: iinc 1, 1
20: goto 2
23: return
```

f()函数，取得一个整数数组，使用的是aload_0 在行偏移3行。取得到了一个数组的大小，等等。

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=1, args_size=1
0: iconst_5
1: newarray int
3: dup
4: iconst_0
5: iconst_1
6: iastore
7: dup
8: iconst_1
9: iconst_2
10: iastore
11: dup
12: iconst_2
13: iconst_3
14: iastore
15: dup
16: iconst_3
17: iconst_4
18: iastore
19: dup
20: iconst_4
21: iconst_5
22: iastore
23: invokestatic #4 // Method f:([I)V
26: return
```

素组在main()函数是构造的，使用newarray指令，被填充慢了之后f()被调用。

随便提一句，数组对象并不是在main()中销毁的，在整个java中也没有被析构。因为JVM的垃圾收集齐不是自动的，当他感觉需要的时候。 format()方法是做什么的？它用两个参数作为输入，字符串和数组对象。

```
public PrintStream format(String format, Object... args⤦ )
```

让我们看一下。

```
public static void main(String[] args)
{
int i=123;
double d=123.456;
System.out.format("int: %d double: %f.%n", i, d⤦
Ç );
}
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=7, locals=4, args_size=1
0: bipush 123
2: istore_1
3: ldc2_w #2 // double 123.456⤦
Ç d
6: dstore_2
7: getstatic #4 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
10: ldc #5 // String int: %d⤦
Ç double: %f.%n
12: iconst_2
13: anewarray #6 // class java/⤦
Ç lang/Object
16: dup
17: iconst_0
18: iload_1
19: invokestatic #7 // Method java/⤦
Ç lang/Integer.valueOf:(I)Ljava/lang/Integer;
22: aastore
23: dup
24: iconst_1
25: dload_2
26: invokestatic #8 // Method java/⤦
Ç lang/Double.valueOf:(D)Ljava/lang/Double;
29: aastore
30: invokevirtual #9 // Method java/io⤦
Ç /PrintStream.format:(Ljava/lang/String;[Ljava/lang/Object⤦
Ç ;)Ljava/io/PrintStream;
33: pop
34: return
```

所以int和double类型是被首先普生为integer和double 对象，被用于方法的值。。。format()方法需要，对象雷翔的对象作为输入，因为integer和double类是继承于根类root。他们适合作为数组输入的元素， 另一方面，数组总是同质的，例如，同一个数组不能含有两种不同的数据类型。不能同时都把integer和double类型的数据同时放入的数组。

数组对象的对象在偏移13行，整型对象被添加到在行偏移22. double对象被添加到数组在29行。

倒数第二的pop指令，丢弃了栈顶的元素，因此，这些return执行，堆栈是的空的（平行）

54.13.6 二位数组

二位数组在java 中是一个数组去引用另外一个数组 让我们来创建二位素组。( )

```
public static void main(String[] args)
{
int[][] a = new int[5][10];
a[1][2]=3;
}
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1
0: iconst_5
1: bipush 10
3: multianewarray #2, 2 // class "[[I"
7: astore_1
8: aload_1
9: iconst_1
10: aaload
11: iconst_2
12: iconst_3
13: iastore
14: return
```

它创建使用的是multianewarry指令：对象类型和维数作为操作数，数组的大小（10*5），返回到栈中。（使用iconst_5和bipush指令）

行引用在行偏移10加载（iconst_1和aaload）列引用是选择使用iconst_2指令，在行偏移11行。值得写入和设定在12行，iastore在13 行，写入数据元素？

```
public static int get12 (int[][] in)
{
return in[1][2];
}
public static int get12(int[][]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: aload_0
1: iconst_1
2: aaload
3: iconst_2
4: iaload
5: ireturn
```

引用数组在行2加载，列的设置是在行3，iaload加载数组。

54.13.7 三维数组

三维数组是，引用一维数组引用一维数组。

```
public static void main(String[] args)
{
int[][][] a = new int[5][10][15];
a[1][2][3]=4;
get_elem(a);
}
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1
0: iconst_5
1: bipush 10
3: bipush 15
5: multianewarray #2, 3 // class "[[[I"
9: astore_1
10: aload_1
11: iconst_1
12: aaload
13: iconst_2
14: aaload
15: iconst_3
16: iconst_4
17: iastore
18: aload_1
19: invokestatic #3 // Method ⤦
Ç get_elem:([[[I)I
22: pop
23: return
```

它是用两个aaload指令去找right引用。

```
public static int get_elem (int[][][] a)
{
return a[1][2][3];
}
public static int get_elem(int[][][]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: aload_0
1: iconst_1
2: aaload
3: iconst_2
4: aaload
5: iconst_3
6: iaload
7: ireturn
```

53.13.8总结

在java中可能出现栈溢出吗？不可能，数组长度实际就代表有多少个对象，数组的边界是可控的，而发生越界访问的情况时，会抛出异常。

54.14 字符串

54.14.1 第一个例子

字符串也是对象，和其他对象的构造方式相同。（还有数组）

```
public static void main(String[] args)
{
System.out.println("What is your name?");
String input = System.console().readLine();
System.out.println("Hello, "+input);
}
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1
0: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
3: ldc #3 // String What is⤦
Ç your name?
5: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
8: invokestatic #5 // Method java/⤦
Ç lang/System.console:()Ljava/io/Console;
11: invokevirtual #6 // Method java/io⤦
Ç /Console.readLine:()Ljava/lang/String;
14: astore_1
15: getstatic #2 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
18: new #7 // class java/⤦
Ç lang/StringBuilder
21: dup
22: invokespecial #8 // Method java/⤦
Ç lang/StringBuilder."<init>":()V
25: ldc #9 // String Hello,
27: invokevirtual #10 // Method java/⤦
Ç lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/⤦
Ç StringBuilder;
30: aload_1
31: invokevirtual #10 // Method java/⤦
Ç lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/⤦
Ç StringBuilder;
34: invokevirtual #11 // Method java/⤦
Ç lang/StringBuilder.toString:()Ljava/lang/String;
37: invokevirtual #4 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
40: return
```

在11行偏移调用了readline()方法，字符串引用（由用户提供）被存储在栈顶，在14行偏移,字符串引用被存储在LVA的1号槽中。

用户输入的字符串在30行偏移处重新加载并和 “hello”字符进行了链接，使用的是StringBulder类，在17行偏移,构造的字符串被pirntln方法打印。

54.14.2 第二个例子

另外一个例子

```
public class strings
{
public static char test (String a)
{
return a.charAt(3);
};
public static String concat (String a, String b)
{
return a+b;
}
}
public static char test(java.lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: aload_0
1: iconst_3
2: invokevirtual #2 // Method java/⤦
Ç lang/String.charAt:(I)C
5: ireturn
```

字符串的链接使用用StringBuilder类完成。

```
public static java.lang.String concat(java.lang.String, java.⤦
Ç lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: new #3 // class java/⤦
Ç lang/StringBuilder
3: dup
4: invokespecial #4 // Method java/⤦
Ç lang/StringBuilder."<init>":()V
7: aload_0
8: invokevirtual #5 // Method java/⤦
Ç lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/⤦
Ç StringBuilder;
11: aload_1
12: invokevirtual #5 // Method java/⤦
Ç lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/⤦
Ç StringBuilder;
15: invokevirtual #6 // Method java/⤦
Ç lang/StringBuilder.toString:()Ljava/lang/String;
18: areturn
```

另外一个例子

```
public static void main(String[] args)
{
String s="Hello!";
int n=123;
System.out.println("s=" + s + " n=" + n);
}
```

字符串构造用StringBuilder类，和它的添加方法，被构造的字符串被传递给println方法。

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=3, args_size=1
0: ldc #2 // String Hello!
2: astore_1
3: bipush 123
5: istore_2
6: getstatic #3 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
9: new #4 // class java/⤦
Ç lang/StringBuilder
12: dup
13: invokespecial #5 // Method java/⤦
Ç lang/StringBuilder."<init>":()V
16: ldc #6 // String s=
18: invokevirtual #7 // Method java/⤦
Ç lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/⤦
Ç StringBuilder;
21: aload_1
22: invokevirtual #7 // Method java/⤦
Ç lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/⤦
Ç StringBuilder;
25: ldc #8 // String n=
27: invokevirtual #7 // Method java/⤦
Ç lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/⤦
Ç StringBuilder;
30: iload_2
31: invokevirtual #9 // Method java/⤦
Ç lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
34: invokevirtual #10 // Method java/⤦
Ç lang/StringBuilder.toString:()Ljava/lang/String;
37: invokevirtual #11 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
40: return
```

54.15 异常

让我们稍微修改一下，月处理的那个例子(在932页的54.13.4)

清单 54.10: IncorrectMonthException.java

```
public class IncorrectMonthException extends Exception
{
private int index;
public IncorrectMonthException(int index)
{
this.index = index;
}
public int getIndex()
{
return index;
}
}
```

清单 54.11: Month2.java

```
class Month2
{
public static String[] months =
{
"January",
"February",
"March",
"April",
"May",
"June",
"July",
"August",
"September",
"October",
"November",
"December"
};
public static String get_month (int i) throws ⤦
Ç IncorrectMonthException
{
if (i<0 || i>11)
throw new IncorrectMonthException(i);
return months[i];
};
public static void main (String[] args)
{
try
{
System.out.println(get_month(100));
}
catch(IncorrectMonthException e)
{
System.out.println("incorrect month ⤦
Ç index: "+ e.getIndex());
e.printStackTrace();
}
};
}
```

本质上，IncorrectMonthExceptinClass类只是做了对象构造，还有访问器方法。IncorrectMonthExceptinClass是继承于Exception类，所以，IncorrectMonth类构造之前，构造父类Exception，然后传递整数给IncorrectMonthException类作为唯一的属性值。

```
public IncorrectMonthException(int);
flags: ACC_PUBLIC
Code:
stack=2, locals=2, args_size=2
0: aload_0
1: invokespecial #1 // Method java/⤦
Ç lang/Exception."<init>":()V
4: aload_0
5: iload_1
6: putfield #2 // Field index:I
9: return
```

getIndex()只是一个访问器，引用到IncorrectMothnException类，被传到LVA的0槽(this指针),用aload_0指令取得， 用getfield指令取得对象的整数值，用ireturn指令将其返回。

```
public int getIndex();
flags: ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
0: aload_0
1: getfield #2 // Field index:I
4: ireturn
```

现在来看下month.class的get_month方法。

清单 54.12: Month2.class

```
public static java.lang.String get_month(int) throws ⤦
Ç IncorrectMonthException;
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=1, args_size=1
0: iload_0
1: iflt 10
4: iload_0
5: bipush 11
7: if_icmple 19
10: new #2 // class ⤦
Ç IncorrectMonthException
13: dup
14: iload_0
15: invokespecial #3 // Method ⤦
Ç IncorrectMonthException."<init>":(I)V
18: athrow
19: getstatic #4 // Field months:[⤦
Ç Ljava/lang/String;
22: iload_0
23: aaload
24: areturn
```

iflt 在行偏移1 ，如果小于的话，

这种情况其实是无效的索引，在行偏移10创建了一个对象，对象类型是作为操作书传递指令的。（这个IncorrectMonthException的构造届时，下标整数是被通过TOS传递的。行15偏移） 时间流程走到了行18偏移，对象已经被构造了，现在athrow指令取得新构对象的引用，然后发信号给JVM去找个合适的异常句柄。

athrow指令在这个不返回到控制流，行19偏移的其他的个基本模块，和异常无关，我们能得到到行7偏移。 句柄怎么工作？ main()在inmonth2.class

清单 54.13: Month2.class

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1
0: getstatic #5 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
3: bipush 100
5: invokestatic #6 // Method ⤦
Ç get_month:(I)Ljava/lang/String;
8: invokevirtual #7 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
11: goto 47
14: astore_1
15: getstatic #5 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
18: new #8 // class java/⤦
Ç lang/StringBuilder
21: dup
22: invokespecial #9 // Method java/⤦
Ç lang/StringBuilder."<init>":()V
25: ldc #10 // String ⤦
Ç incorrect month index:
27: invokevirtual #11 // Method java/⤦
Ç lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/⤦
Ç StringBuilder;
30: aload_1
31: invokevirtual #12 // Method ⤦
Ç IncorrectMonthException.getIndex:()I
34: invokevirtual #13 // Method java/⤦
Ç lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
37: invokevirtual #14 // Method java/⤦
Ç lang/StringBuilder.toString:()Ljava/lang/String;
40: invokevirtual #7 // Method java/io⤦
Ç /PrintStream.println:(Ljava/lang/String;)V
43: aload_1
44: invokevirtual #15 // Method ⤦
Ç IncorrectMonthException.printStackTrace:()V
47: return
Exception table:
from to target type
0 11 14 Class IncorrectMonthException
```

这是一个异常表，在行偏移0-11（包括）行，一个IncorrectinMonthException异常可能发生，如果发生，控制流到达14行偏移，确实main程序在11行偏移结束，在14行异常开始， 没有进入此区域条件(condition/uncondition)设定，是不可能到打这个位置的。（PS：就是没有异常捕获的设定，就不会有异常流被调用执行。）

但是JVM会传递并覆盖执行这个异常case。 第一个astore_1(在行偏移14)取得，将到来的异常对象的引用，存储在LVA的槽参数1之后。getIndex()方法（这个异常对象） 会被在31行偏移调用。引用当前的异常对象，是在30行偏移之前。 所有的这些代码重置都是字符串操作代码：第一个整数值使用的是getIndex()方法，被转换

成字符串使用的是toString()方法，它会和“正确月份下标”的文本字符来链接（像我们之前考虑的那样）。println()和printStackTrace(1)会被调用，PrintStackTrace(1) 调用 结束之后，异常被捕获，我们可以处理正常的函数，在47行偏移，return结束main（）函数，如果没有发生异常，不会执行任何的代码。

这有个例子，IDA是如何显示异常范围：

清单54.14 我从我的计算机中找到 random.class 这个文件

```
.catch java/io/FileNotFoundException from met001_335 to ⤦
Ç met001_360\
using met001_360
.catch java/io/FileNotFoundException from met001_185 to ⤦
Ç met001_214\
using met001_214
.catch java/io/FileNotFoundException from met001_181 to ⤦
Ç met001_192\
using met001_195
951
CHAPTER 54. JAVA 54.16. CLASSES
.catch java/io/FileNotFoundException from met001_155 to ⤦
Ç met001_176\
using met001_176
.catch java/io/FileNotFoundException from met001_83 to ⤦
Ç met001_129 using \
met001_129
.catch java/io/FileNotFoundException from met001_42 to ⤦
Ç met001_66 using \
met001_69
.catch java/io/FileNotFoundException from met001_begin to ⤦
Ç met001_37\
using met001_37
```

〔校准到这结束。〕

54.16 类

简单类

清单 54.15: test.java

```
public class test
{
public static int a;
private static int b;
public test()
{
a=0;
b=0;
}
public static void set_a (int input)
{
a=input;
}
public static int get_a ()
{
return a;
}
public static void set_b (int input)
{
b=input;
}
public static int get_b ()
{
return b;
}
}
```

构造函数，只是把两个之段设置成0.

```
public test();
flags: ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
0: aload_0
1: invokespecial #1 // Method java/⤦
Ç lang/Object."<init>":()V
4: iconst_0
5: putstatic #2 // Field a:I
8: iconst_0
9: putstatic #3 // Field b:I
12: return
```

a的设定器

```
public static void set_a(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iload_0
1: putstatic #2 // Field a:I
4: return
```

a的取得器

```
public static int get_a();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=0, args_size=0
0: getstatic #2 // Field a:I
3: ireturn
```

b的设定器

```
public static void set_b(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iload_0
1: putstatic #3 // Field b:I
4: return
```

b的取得器

```
public static int get_b();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=0, args_size=0
0: getstatic #3 // Field b:I
3: ireturn
```

类中的公有和私有字段代码没什么区别。但是类型信息会在in.class 文件中表示，并且，无论如何私有变量是不可以被访问的。

让我们创建对象并调用方法： 清单 54.16: ex1.java

新指令创建对象，但不调用构造函数（它在4行偏移被调用）set_a()方法被在16行偏移被调用，字段访问使用的getstatic指令,在行偏移21。

```
Listing 54.16: ex1.java
public class ex1
{
public static void main(String[] args)
{
test obj=new test();
obj.set_a (1234);
System.out.println(obj.a);
}
}
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=1
0: new #2 // class test
3: dup
4: invokespecial #3 // Method test."<⤦
Ç init>":()V
7: astore_1
8: aload_1
9: pop
10: sipush 1234
13: invokestatic #4 // Method test.⤦
Ç set_a:(I)V
16: getstatic #5 // Field java/⤦
Ç lang/System.out:Ljava/io/PrintStream;
19: aload_1
20: pop
21: getstatic #6 // Field test.a:I
24: invokevirtual #7 // Method java/io⤦
Ç /PrintStream.println:(I)V
27: return
```

54.17 简单的补丁。

54.17.1 第一个例子

让我们进入一个简单的修补任务。

```
public class nag
{
public static void nag_screen()
{
System.out.println("This program is not ⤦
Ç registered");
};
public static void main(String[] args)
{
System.out.println("Greetings from the mega-⤦
Ç software");
nag_screen();
}
}
```

我们如何去除"This program is registered"的打印输出.

最会在IDA中加载.class文件。

清单54.1: IDA

我们修补一下函数的第一个byte在177(返回指令操作码)

Figure 54.2 : IDA

这个在JDK1.7中不工作

```
Exception in thread "main" java.lang.VerifyError: Expecting a ⤦
Ç stack map frame
Exception Details:
Location:
nag.nag_screen()V @1: nop
Reason:
Error exists in the bytecode
Bytecode:
0000000: b100 0212 03b6 0004 b1
at java.lang.Class.getDeclaredMethods0(Native Method)
at java.lang.Class.privateGetDeclaredMethods(Class.java⤦
Ç :2615)
at java.lang.Class.getMethod0(Class.java:2856)
at java.lang.Class.getMethod(Class.java:1668)
at sun.launcher.LauncherHelper.getMainMethod(⤦
Ç LauncherHelper.java:494)
at sun.launcher.LauncherHelper.checkAndLoadMain(⤦
Ç LauncherHelper.java:486)
```

也许，JVM有一些其他检查，关联到栈映射。 好的，我们修补成不同的，去掉nag()函数调用。

清单:54.5 IDA NOP的操作码是0: 这个可以了！

54.17.2第二个例子

现在是另外一个简单的crackme例子。

```
public class password
{
public static void main(String[] args)
{
System.out.println("Please enter the password")⤦
Ç ;
String input = System.console().readLine();
if (input.equals("secret"))
System.out.println("password is correct⤦
Ç ");
957
CHAPTER 54. JAVA 54.17. SIMPLE PATCHING
else
System.out.println("password is not ⤦
Ç correct");
}
}
```

图54.4:IDA 我们看ifeq指令是怎么工作的，他的名字的意思是如果等于。 这是不恰当的，我更愿意命名if (ifz if zero) 如果栈顶值是0，他就会跳转，在我们这个例子，如果密码 不正确他就跳转。（equal方法返回的是0） 首先第一个方案就是修该这个指令... iefq是两个bytes的操作码 编码和跳转偏移，让这个指令定制，我们必须设定byte3 3byte（因为3是要添加当前地址结果，总是跳转同下一条指令） 因为ifeq的指令长度就是3bytes.

图54.5IDA

这个在JDK1.7中不工作

```
Exception in thread "main" java.lang.VerifyError: Expecting a ↙
Ç stackmap frame at branch target 24
Exception Details:
Location:
password.main([Ljava/lang/String;)V @21: ifeq
Reason:
Expected stackmap frame at this location.
Bytecode:
0000000: b200 0212 03b6 0004 b800 05b6 0006 4c2b
0000010: 1207 b600 0899 0003 b200 0212 09b6 0004
0000020: a700 0bb2 0002 120a b600 04b1
Stackmap Table:
append_frame(@35,Object[#20])
same_frame(@43)
at java.lang.Class.getDeclaredMethods0(Native Method)
at java.lang.Class.privateGetDeclaredMethods(Class.java↙
Ç :2615)
at java.lang.Class.getMethod0(Class.java:2856)
at java.lang.Class.getMethod(Class.java:1668)
at sun.launcher.LauncherHelper.getMainMethod(↙
Ç LauncherHelper.java:494)
959
CHAPTER 54. JAVA 54.18. SUMMARY
at sun.launcher.LauncherHelper.checkAndLoadMain(↙
Ç LauncherHelper.java:486)
```

不用说了，它工作在JRE1.6 我也尝试把所有的3 ifeq的所有操作码都用0替换（NOP），它仍然会工作，好，可能没有更多的堆栈映射在JRE1.7中被检查出来。

好的，我替换整个equal方法调用，使用icore_1指令加NOPS的修改。

（TOS）栈顶总是1，当ifeq指令被执行...所以ifeq也不会被执行。

可以了。

54.18总结

和C/C+比较java少了一些什么？

- 结构体：使用类
- 联合：使用类继承。
- 无符号数据类型，多说一句，还有一些在Java中实现的加密算法的硬编码。
- 函数指针。

Part V 在代码里面寻找重要又有趣的东西

PART V 寻找代码中有趣或者重要的部分

现代软件设计中，极简不是特别重要的特性。

并不是因为程序员编写的代码多，而是由于许多库通常都会静态链接到可执行文件中。如果所有的外部库都移入了外部DLL文件中，情况将有所不同。(C++使用STL和其他模版库的另一个原因)

因此，确定函数的来源很重要，是否来源于标准库或者其他著名的库(比如Boost，libpng)，是否与我们在代码中寻找的东西相关。

通过重写所有的C/C++代码来寻找我们想要的东西是不现实的。

逆向工程师的一个主要的任务是迅速定位到目标代码。

IDA反汇编工具允许我们搜索文本字符串，字节序列和常量。甚至可以导出为.lst或者.asm文件，然后使用grep,awk等工具进一步分析。

当你尝试去理解某些代码的功能时，一些开源库比如libpng会容易理解一些。当你觉得某些常量或者文本字符串眼熟时，值得用google搜索一下。如果你发现他们在某些地方使用了开源项目时，那么只要对比一下函数就可以了。这些方法能够解决部分问题。

举个例子，如果一个程序使用XML文件，那么第一步是确定使用了哪个XML库。通常情况下使用的是标准库(或者有名的库)而非自编写的库。

再举个例子，有一次我尝试去理解SAP 6.0中网络包如何压缩与解压。整个软件很大，但手头有一个包含详细debug信息的.PDB文件，非常方便。最后我找到一个负责解压网络包的函数，叫CsDecomprLZC。我马上就用google搜索了函数名，发现MaxDB(一个开源SAP项目)也使用了这个函数。http://www.google.com/search?q=CsDecomprLZC

然后惊奇的发现，MaxDB和SAP 6.0 使用同样的代码来处理压缩和解压网络包。

第55章

识别可执行文件

55.1 Microsoft Visual C++

可导入的MSVC版本和DLL文件如下图：

Marketing version	Internal version	CL.EXE version	DLLs that can be imported	Release date
6	6.0	12.00	msvcrt.dll, msvcp60.dll	June 1998
.NET (2002)	7.0	13.00	msvcr70.dll, msvcp70.dll	February 13, 2002
.NET 2003	7.1	13.10	msvcr71.dll, msvcp71.dll	April 24, 2003
2005	8.0	14.00	msvcr80.dll, msvcp80.dll	November 7, 2005
2008	9.0	15.00	msvcr90.dll, msvcp90.dll	November 19, 2007
2010	10.0	16.00	msvcr100.dll, msvcp100.dll	April 12, 2010
2012	11.0	17.00	msvcr110.dll, msvcp110.dll	September 12, 2012
2013	12.0	18.00	msvcr120.dll, msvcp120.dll	October 17, 2013

msvcp*.dll包含C++相关函数，因此如果导入了这类dll，便可推测是C++程序。

55.1.1命名管理

命名通常以问号?开始。

获取更多关于MSVC命令管理的信息：51.1.1节

55.2 GCC

除了*NIX环境，Win32下也有GCC，需要Cygwin和MinGW。

55.2.1 命名管理

命名通常以_Z符号开头。

更多关于GCC命名管理的信息：51.1.1节

55.2.2 Cygwin

cygwin1.dll经常被导入。

55.2.3 MinGW

msvcrt.dll可能会被导入。

55.3 Intel FORTRAN

libifcoremd.dll,libifportmd.dll和libiomp5md.dll(OpenMP支持)可能会被导入。

libifcoremd.dll中许多函数以前缀名for_开始，表示FORTRAN。

55.4Watcom,OpenWatcom

55.4.1 命名管理

命名通常以W符号开始。

举个例子，下面是"class"类名为"method"的方法没有任何参数并且返回void的加密：

```
W?method$_class$n__v
```

55.5 Borland

这里有一个有关Borland Delphi和C++开发者命名管理的例子：

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiiit1iiiiii
@TrueColorTo16BitN$qpviiiiiit1iiiiii
@DIB24BitTo8BitBitmap$qpviiiiiit1iiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
@TrueBitmap@$bctr$qiilll
```

命名通常以@符号开始，然后是类名、方法名、加密方法的参数类型。

这些名称会被导入到.exe，.dll和debug信息内等等。

Borland Visual Component Libarary(VCL)存储在.bpl文件中，而不是.dll。比如vcl50.dll,rtl60.dll。

其他可能导入的DLL：BORLNDMM.DLL。

55.5.1 Delphi

几乎所有的Delphi可执行文件的代码段都以"Boolean"字符串开始，和其他类型名称一起。 下面是一个典型的Delphi程序的代码段开头，这个块紧接着win32 PE文件头：

```
00000400  04 10 40 00 03 07 42 6f  6f 6c 65 61 6e 01 00 00  |..@
...Boolean...|
00000410  00 00 01 00 00 00 00 10  40 00 05 46 61 6c 73 65  |...
.....@..False|
00000420  04 54 72 75 65 8d 40 00  2c 10 40 00 09 08 57 69  |.Tr
ue.@.,.@...Wi|
00000430  64 65 43 68 61 72 03 00  00 00 00 ff ff 00 00 90  |deC
har.........|
00000440  44 10 40 00 02 04 43 68  61 72 01 00 00 00 00 ff  |D.@
```

```
...Char......|
00000450  00 00 00 90 58 10 40 00  01 08 53 6d 61 6c 6c 69  |...
.X.@...Smalli|
00000460  6e 74 02 00 80 ff ff ff  7f 00 00 90 70 10 40 00  |nt.
.........p.@.|
00000470  01 07 49 6e 74 65 67 65  72 04 00 00 00 80 ff ff  |..I
nteger.......|
00000480  ff 7f 8b c0 88 10 40 00  01 04 42 79 74 65 01 00  |...
...@...Byte..|
00000490  00 00 00 ff 00 00 00 90  9c 10 40 00 01 04 57 6f  |...
.......@...Wo|
000004a0  72 64 03 00 00 00 00 ff  ff 00 00 90 b0 10 40 00  |rd.
...........@.|
000004b0  01 08 43 61 72 64 69 6e  61 6c 05 00 00 00 00 ff  |..C
ardinal......|
000004c0  ff ff ff 90 c8 10 40 00  10 05 49 6e 74 36 34 00  |...
...@...Int64.|
000004d0  00 00 00 00 00 00 80 ff  ff ff ff ff ff ff 7f 90  |...
.............|

000004e0  e4 10 40 00 04 08 45 78  74 65 6e 64 65 64 02 90  |..@
...Extended..|
000004f0  f4 10 40 00 04 06 44 6f  75 62 6c 65 01 8d 40 00  |..@
...Double..@.|
00000500  04 11 40 00 04 08 43 75  72 72 65 6e 63 79 04 90  |..@
...Currency..|
00000510  14 11 40 00 0a 06 73 74  72 69 6e 67 20 11 40 00  |..@
...string .@.|
00000520  0b 0a 57 69 64 65 53 74  72 69 6e 67 30 11 40 00  |..W
ideString0.@.|
00000530  0c 07 56 61 72 69 61 6e  74 8d 40 00 40 11 40 00  |..V
ariant.@.@.@.|
00000540  0c 0a 4f 6c 65 56 61 72  69 61 6e 74 98 11 40 00  |..O
leVariant..@.|
00000550  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |...
.............|
00000560  00 00 00 00 00 00 00 00  00 00 00 00 98 11 40 00  |...
...........@.|
00000570  04 00 00 00 00 00 00 00  18 4d 40 00 24 4d 40 00  |...
......M@.$M@.|
00000580  28 4d 40 00 2c 4d 40 00  20 4d 40 00 68 4a 40 00  |(M@
.,M@. M@.hJ@.|
00000590  84 4a 40 00 c0 4a 40 00  07 54 4f 62 6a 65 63 74  |.J@
..J@..TObject|
000005a0  a4 11 40 00 07 07 54 4f  62 6a 65 63 74 98 11 40  |..@
...TObject..@|
000005b0  00 00 00 00 00 00 00 06  53 79 73 74 65 6d 00 00  |...
.....System..|
000005c0  c4 11 40 00 0f 0a 49 49  6e 74 65 72 66 61 63 65  |..@
...IInterface|
000005d0  00 00 00 00 01 00 00 00  00 00 00 00 00 c0 00 00  |...
.............|
000005e0  00 00 00 00 46 06 53 79  73 74 65 6d 03 00 ff ff  |...
```

```
.F.System....|
000005f0  f4 11 40 00 0f 09 49 44  69 73 70 61 74 63 68 c0  |..@
...IDispatch.|
00000600  11 40 00 01 00 04 02 00  00 00 00 00 c0 00 00 00  |.@.
.............|
00000610  00 00 00 46 06 53 79 73  74 65 6d 04 00 ff ff 90  |...
F.System.....|
00000620  cc 83 44 24 04 f8 e9 51  6c 00 00 83 44 24 04 f8  |..D
$...Ql...D$..|
00000630  e9 6f 6c 00 00 83 44 24  04 f8 e9 79 6c 00 00 cc  |.ol
...D$...yl...|
00000640  cc 21 12 40 00 2b 12 40  00 35 12 40 00 01 00 00  |.!.
@.+.@.5.@....|
00000650  00 00 00 00 00 00 00 00  00 c0 00 00 00 00 00 00  |...
.............|
00000660  46 41 12 40 00 08 00 00  00 00 00 00 00 8d 40 00  |FA.
@..........@.|
00000670  bc 12 40 00 4d 12 40 00  00 00 00 00 00 00 00 00  |..@
.M.@.........|
00000680  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |...
.............|
00000690  bc 12 40 00 0c 00 00 00  4c 11 40 00 18 4d 40 00  |..@
.....L.@..M@.|
000006a0  50 7e 40 00 5c 7e 40 00  2c 4d 40 00 20 4d 40 00  |P~@
.\~@.,M@. M@.|
000006b0  6c 7e 40 00 84 4a 40 00  c0 4a 40 00 11 54 49 6e  |l~@
..J@..J@..TIn|
000006c0  74 65 72 66 61 63 65 64  4f 62 6a 65 63 74 8b c0  |ter
facedObject..|
000006d0  d4 12 40 00 07 11 54 49  6e 74 65 72 66 61 63 65  |..@
...TInterface|
000006e0  64 4f 62 6a 65 63 74 bc  12 40 00 a0 11 40 00 00  |dOb
ject..@...@..|
000006f0  00 06 53 79 73 74 65 6d  00 00 8b c0 00 13 40 00  |..S
ystem......@.|
00000700  11 0b 54 42 6f 75 6e 64  41 72 72 61 79 04 00 00  |..T
BoundArray...|
00000710  00 00 00 00 00 03 00 00  00 6c 10 40 00 06 53 79  |...
......l.@..Sy|
00000720  73 74 65 6d 28 13 40 00  04 09 54 44 61 74 65 54  |ste
m(.@...TDateT|
00000730  69 6d 65 01 ff 25 48 e0  c4 00 8b c0 ff 25 44 e0  |ime
..%H......%D.|
```

数据段(DATA)最开始的四字节可能是00 00 00 00，32 13 8B C0或者FF FF FF FF。在处理加壳/加密的 Delphi可执行文件时这个信息很有用。

55.6其他有名的DLLs

- vcomp*.dll Microsoft实现的OpenMP

第56章

与外部世界通信(win32)

有时理解函数的功能通过观察函数的输入与输出就足够了。这样可以节省时间。

文件和注册访问：对于最基本的分析，SysInternals的Process Monitor工具很有用。

对于基本网络访问分析，Wireshark很有帮助。

但接下来你仍需查看内部。

第一步是查看使用的是OS的API哪个函数，标准库是什么。

如果程序被分为主要的可执行文件和一系列DLL文件，那么DLL文件中的函数名可能会有帮助。

如果我们对指定文本调用MessageBox()的细节感兴趣，我们可以在数据段中查找这个文本，定位文本引用处，以及控制权交给我们感兴趣的MessageBox()的地方。

如果我们在谈论电子游戏，并且对里面的事件的随机性感兴趣，那么我们可以查找rand()函数或者类似函数(比如马特赛特旋转演算法)，然后定位调用这些函数的地方，更重要的是，函数执行结果如何被使用。

但如果不是一个游戏，并且仍然使用了rand()函数，找出原因也很有意思。这里有一些关于在数据压缩算法中意外出现rand()函数调用的例子(模仿加密)：blog.yurichev.com

56.1 Windows API中常用的函数

下面这些函数可能会被导入。值得注意的是并不是每个函数都在代码中使用。许多函数可能被库函数和CRT代码调用。

- 注册表访问(advapi32.dll):RegEnumKeyEx, RegEnumValue, RegGetValue7, RegOpenKeyEx, RegQueryValueEx
- .ini-file访问(kernel32.dll): GetPrivateProfileString
- 资源访问(68.2.8): (user32.dll): LoadMen
- TCP/IP网络(ws2_32.dll): WSARecv, WSASend
- 文件访问(kernel32.dll): CreateFile, ReadFile, ReadFileEx, WriteFile, WriteFileEx
- Internet高级访问(wininet.dll): WinHttpOpen
- 可执行文件数字签名(wintrust.dll): WinVerifyTrust
- 标准MSVC库(如果是动态链接的) (msvcr*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr,

strchr

56.2 tracer:拦截所有函数特殊模块

这里有一个INT3断点，只触发了一次，但可以为指定DLL中的所有函数设置。

```
--one-time-INT3-bp:somedll.dll!.*
```

我们给所有前缀是xml的函数设置INT3断点吧：

```
--one-time-INT3-bp:somedll.dll!xml.*
```

另一方面，这样的断点只会触发一次。

Tracer会在函数调用发生时显示调用情况，但只有一次。但查看函数参数是不可能的。

尽管如此，在你知道这个程序使用了一个DLL，但不知道实际上使用了哪个函数并且有许多的函数的情况下，这个特性还是很有用的。

举个例子，我们来看看，cygwin的uptime工具使用了什么：

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

我们可以看见所有的至少调用了一次的cygwin1.dll库函数，以及位置：

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime
.exe!OEP+0x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from up
time.exe!OEP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from upt
ime.exe!OEP+0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from up
time.exe!OEP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from upt
ime.exe!OEP+0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptim
e.exe!OEP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from upt
ime.exe!OEP+0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptim
e.exe!OEP+0x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from upti
me.exe!OEP+0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.e
xe!OEP+0x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime
.exe!OEP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.
exe!OEP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.e
xe!OEP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from upt
ime.exe!OEP+0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptim
e.exe!OEP+0x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from upti
me.exe!OEP+0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from upti
me.exe!OEP+0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from upti
me.exe!OEP+0x3e6 (0x4013
```

第57章

字符串

57.1 文本字符串

57.1.1 C/C++

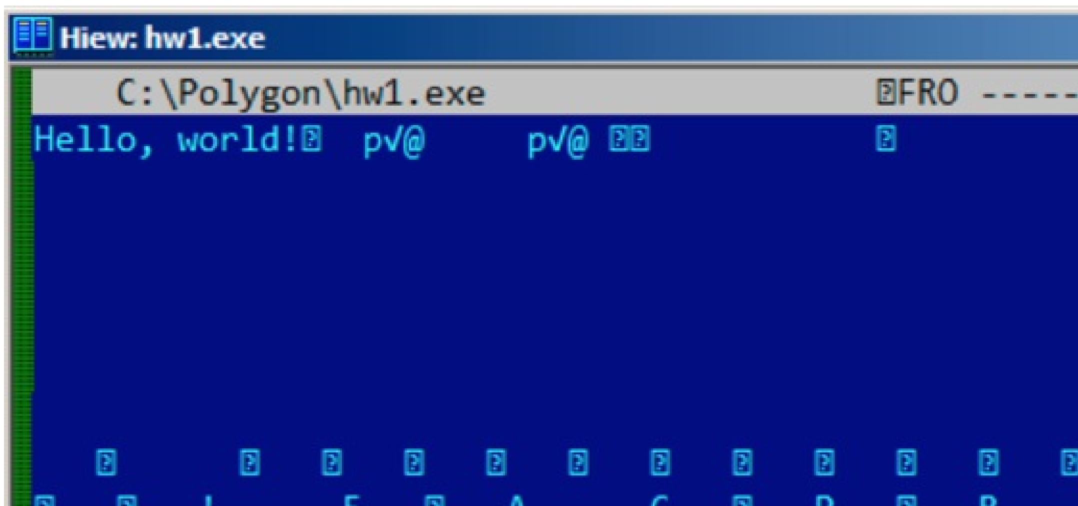
普通的C字符串是以零结束的(ASCIIZ字符串)。

C字符串格式(以零结束)是这样的是出于历史原因。[Rit79中]:

```
A minor difference was that the unit of I/O was the word, not th
e byte, because the PDP-7 was a word- addressed machine. In prac
tice this meant merely that all programs dealing with character
streams ignored null characters, because null was used to pad a
file to an even number of characters.
```

在Hiew或者FAR Manager中，这些字符串看上去是这样的：

```
int main() {
printf ("Hello, world!\n");
};
```



57.1.2 Borland Delphi

在Pascal和Borland Delphi中字符串为8-bit或者32-bit长。

举个例子：

```
CODE:00518AC8                 dd 19h
CODE:00518ACC aLoading___Plea db 'Loading... , please wait.',0
...
CODE:00518AFC                 dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run...',0
```

57.1.3 Unicode

通常情况下，称Unicode是一种编码字符串的方法，每个字符占用2个字节或者16bit。这是一种常见的术语错误。在许多语言系统中，Unicode是一种用于给每个字符分配数字的标准，而不是用于描述编码的方法。

最常用的编码方法是：UTF-8(在Internet和*NIX系统中使用较多)和UTF-16LE(在Windows中使用)。

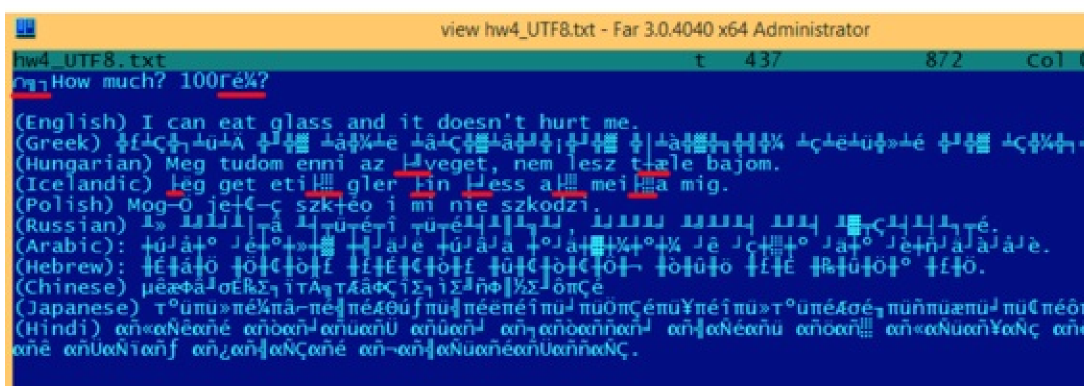
UTF-8

UTF-8是最成功的字符编码方法之一。所有拉丁符号编码成ASCII，而超出ASCII表的字符的编码使用多个字节。0的编码方式和以前一样，所有的标准C字符串函数处理UTF-8字符串和处理其他字符串一样。

我们来看看不同语言中的符号在UTF-8中是如何被编码的，在FAR中看上去又是什么样的，使用437内码表:

```
How much? 100€?

(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic): أنا قادر على أكل الزجاج و هذا لا يؤلمني.
(Hebrew): אני יכול לאכול זכוכית וזה לא מזיק לי.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```



就像你看到的一样，英语字符串看上去和ASCII编码的一样。匈牙利语使用一些拉丁符号加上变音标志。这些符号使用多个字节编码。我用红色下划线标记出来了。对于冰岛语和波兰语也是一样的。我在开始处使用"Euro"通行符号，编码为3个字

节。这里剩下的语言系统与拉丁文没有联系。至少在俄语、阿拉伯语、希伯来语和印地语中我们可以看到相同的字节，这并不稀奇：语言系统的所有符号通常位于同一个Unicode表中，所以他们的号码前几个数字相同。

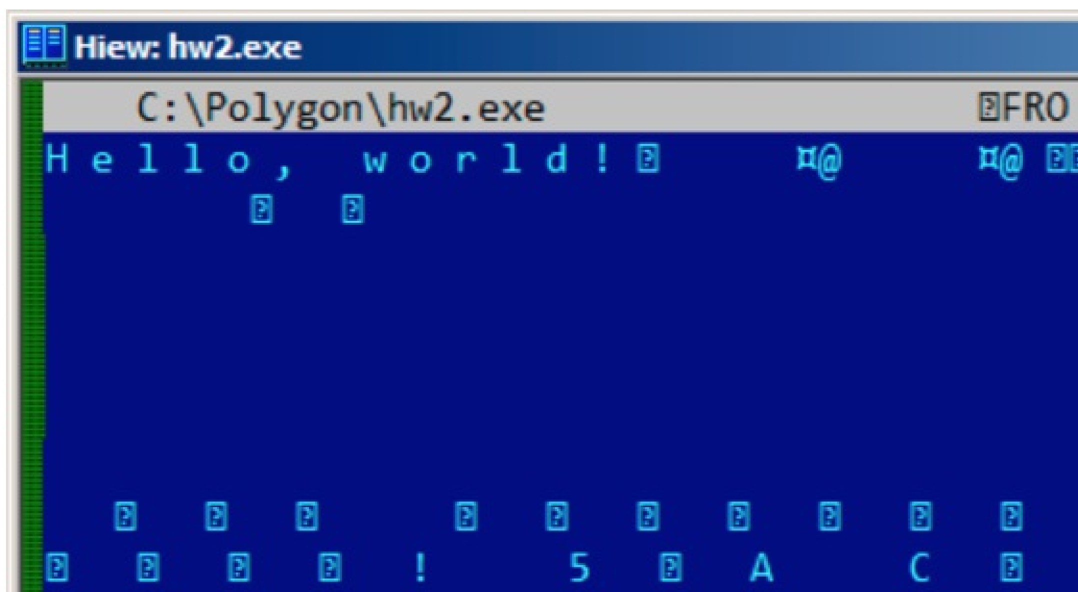
之前在"How much?"前面，我们看到了3个字节，这实际上是BOM。BOM定义了使用的编码系统。

UTF-16LE

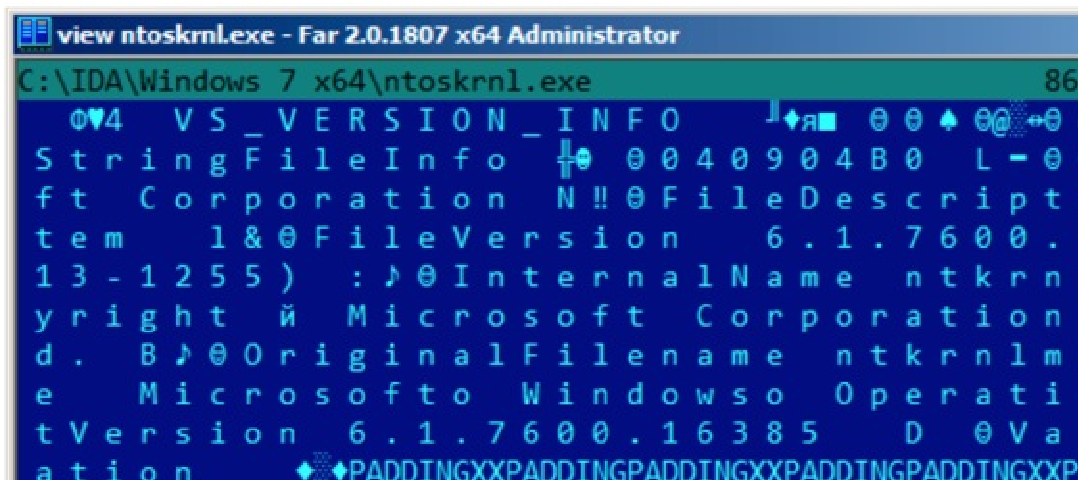
在Windows中，许多win32函数带有后缀 -A和-W。第一种类型的函数用于处理普通字符串，第二种类型的函数用于处理UTF-16LE(wide)，每个符号存储类型通常为16比特的short。

UTF-16中拉丁符号在Hiew和FAR中看上去插入了0字节：

```
int wmain() {
wprintf (L"Hello, world!\n");
};
```



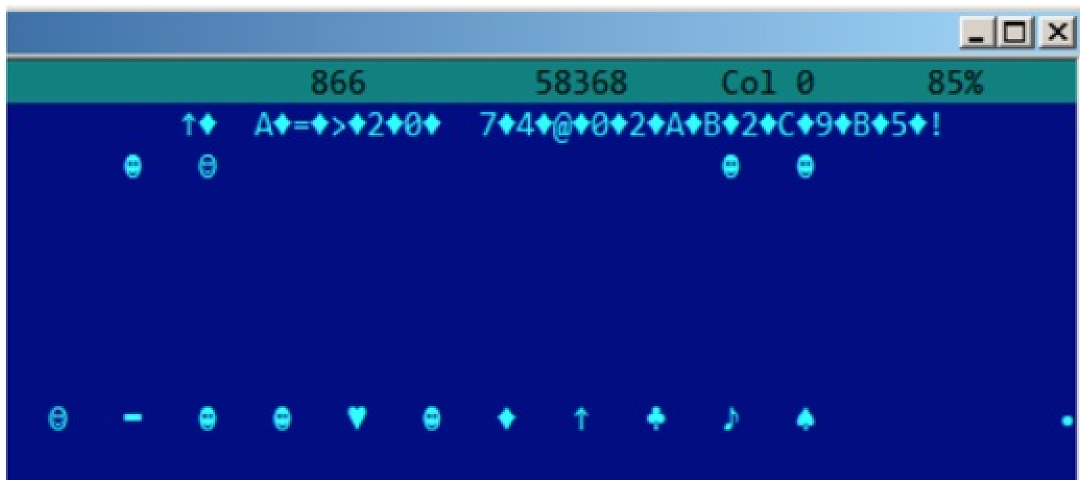
在Windows NT系统中经常可以看见这样的：



在IDA中，占两个字节通常被称为Unicode：

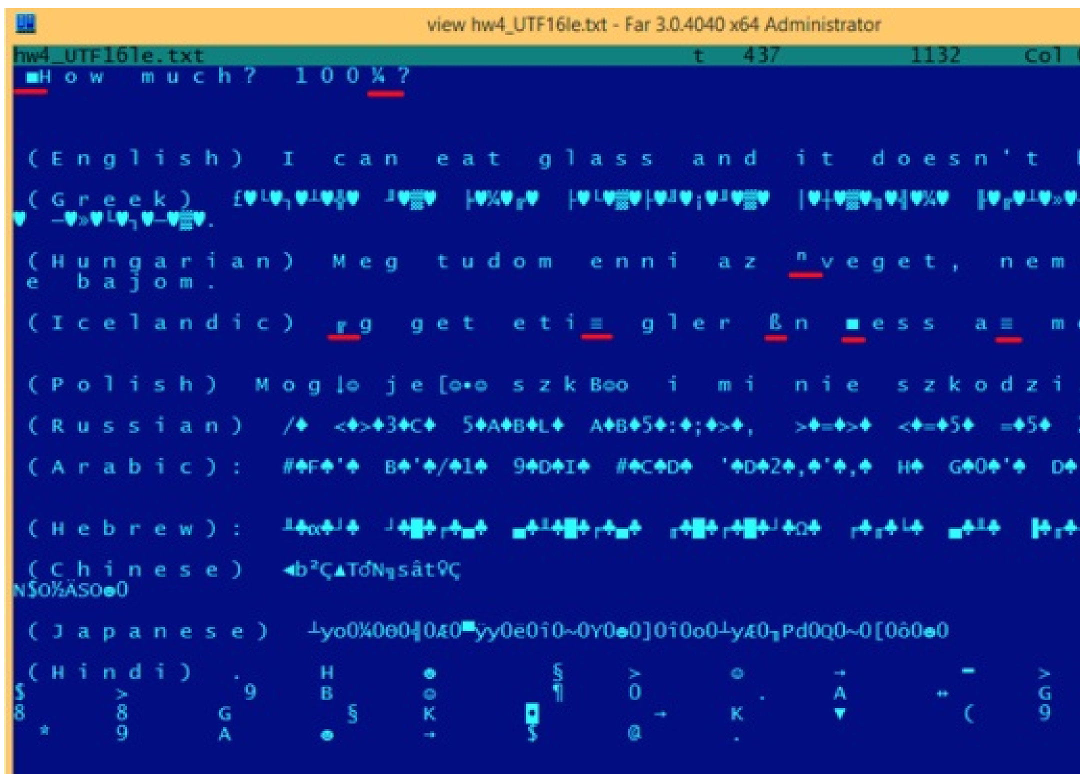
```
.data:0040E000 aHelloWorld:
.data:0040E000                  unicode 0, <Hello, world!>
.data:0040E000                  dw 0Ah, 0
```

下面是俄语字符串在UTF-16LE中如何被编码：



容易发现的是，符号被插入了方形字符(ASCII码为4).实际上，西里尔符号位于Unicode第四个平面。因此，在UTF-16LE中，西里尔符号的范围为0x400到0x4FF.

我们回到使用多种语言书写的字符串的例子中吧。下面是他们在UTF-16LE中的样子。



这里我们也能看到开始处有一个BOM。所有的拉丁字符都被插入了一个0字节。我也给一些带有变音符号的字符标注了红色下划线(匈牙利语和冰岛语)。

57.1.4 Base64

Base64编码方法多用于需要将二进制数据以文本字符串的形式传输的情况。实际上，这种算法将3个二进制字节编码为4个可打印字符：所有拉丁字母(包括大小写)、数字、加号、除号共64个字符。

Base64字符串一个显著的特性是他们经常(并不总是)以1个或者2个等号结尾，举个例子：


```
AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp
9lAOuWs=
```



```
WVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp
9lAOuQ==
```


等号不会在base-64编码的字符串中间出现。

57.2 Error/debug messages

调试信息非常有帮助。在某种程度上，调试信息报告了程序当前的行为。通常这些printf类函数，写入信息到log文件中，在release模式下不写任何东西但会显示调用信息。如果本地或全局变量dump到了调试信息中，可能会有帮助，至少能获取变量名。比如在Oracle RDBMS中就有这样一个函数 ksdewt()。

文本字符串常常很有帮助。IDA反汇编器可以展示指定字符串被哪个函数在哪里使用。经常会出现有趣的状况。

错误信息也很有帮助。在Oracle RDBMS中，错误信息会报告使用的一系列函数。

更多相关信息：blog.yurichev.com。

快速获知哪个函数在什么情况下报告了错误信息是可以做到的。顺便说一句，这也是copy-protection系统为什么要设置模糊而难懂的错误信息或错误码。没有人会为软件破解者仅仅通过错误信息就快速找到了copy-protection被触发的原因而感到高兴。

一个关于错误信息编码的例子：78.2节

57.3 Suspicious magic strings

一些幻数字符串通常使用在后门中，看上去很神秘。举个例子，下面有一个TP-Link WR740路由器的后门。使用下面的URL可以激活后门：http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html。

实际上，"userRpmNatDebugRpm26525557"字符串会在硬件中显示。在后门信息泄漏前，这个字符串并不能被google到。你在任何RFC中都找不到这个。你也无法在任何计算机科学算法中找到使用了这个奇怪字节序列的地方。此外，这看上去也不像错误信息或者调试信息。因此，调查这样一个奇怪字符串的用途是明智的。

有时像这样的字符串可能使用了base64编码。所以解码后再看一遍是明智的，甚至扫一眼就够了。

更确切的说，这种隐藏后门的方法称为“security through obscurity”。

第58章

调用assert

有时，assert()宏的出现也是有用的：通常这个宏会泄漏源文件名，行号和条件。

最有用的信息包含在assert的条件中，我们可以从中推断出变量名或者结构体名。另一个有用的信息是文件名。我们可以从中推断出使用了什么类型的代码。并且也可能通过文件名识别出有名的开源库。

```
.text:107D4B29 mov  dx, [ecx+42h]
.text:107D4B2D cmp  edx, 1
.text:107D4B30 jz   short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ; "td->td_planarconfi
g == PLANARCONFIG_CON"...
.text:107D4B41 call ds:_assert
...
.text:107D52CA mov  edx, [ebp-4]
.text:107D52CD and  edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz   short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30     ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert
...
.text:107D6759 mov  cx, [eax+6]
.text:107D675D cmp  ecx, 0Ch
.text:107D6760 jle  short loc_107D677A
.text:107D6762 push 2D8h
.text:107D6767 push offset aLzw_c   ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= B
ITS_MAX"
.text:107D6771 call ds:_assert
```

同时google一下条件和文件名是明智的，可能会因此找到开源库。举个例子，如果我们google查找“sp->lzw_nbits <= BITS_MAX”，将会显示一些与LZW压缩有关的开源代码。

第59章

常量

通常人们在生活中或者程序员在编写代码时喜欢使用像10，100，1000这样的整数。

有经验的逆向工程师会对这些数字的十六进制形式很熟悉：10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710。

常量 0xAAAAAAAA (10101010101010101010101010101010)和0x55555555 (01010101010101010101010101010101)也很常用——构成alternating bits。举个例子，0x55AA在引导扇区，MBR，IBM兼容扩展卡中使用过。

某些算法，特别是密码学方面的使用的常量很有代表性，我们可以在IDA中轻松找到。

举个例子，MD5算法这样初始化内部变量：

var int h0 := 0x67452301

var int h1 := 0xEFCDAB89

var int h2 := 0x98BADCFE

var int h3 := 0x10325476

如果你在代码中某行发现这四个常量，那么极有可能该处函数与MD5有关。

另一个有关CRC16/CRC32算法的例子，通常使用预先计算好的表来计算：

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 +
x^2 + 1) */
u16 const crc16_table[256] = {
        0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280,
0xC241,
        0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481,
0x0440,
        0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81,
0x0E40,
        ...
```

CRC3预计算表同见：第37节

59.1 幻数

许多文件格式定义了标准的文件头，使用了幻数。

举个例子，所有的Win32和MS-DOS可执行文件以"MZ"这两个字符开始。

MIDI文件的开始有"MThd"标志。如果我们有一个使用MIDI文件的程序，它很有可能会检查至少4字节的文件头来确认文件类型。

可以这样实现：

(buf指向内存文件加载的开始处)

```
cmp [buf], 0x6468544D ; "MThd"

jnz _error_not_a_MIDI_file
```

也可能会调用某个函数比如memcmp()或者等同于CMPSB指令(A.6.3节)的代码用于比对内存块。

当你发现这样的地方，你就可以确定的MIDI文件加载的开始处，同时我们可以看到缓冲区存放MIDI文件内容的地方，什么内容被使用以及如何使用。

59.1.1 Dates

59.1.2 DHCP

上面的方法对于网络协议也同样适用。举个例子，DHCP协议网络包包含了magic cookie：0x6353826。任何生成DHCP包的代码在某处一定将这个常量嵌入了包中。它在代码中出现的地方可能就与执行这些操作有关，或者不仅是如此。任何接收DHCP的包都会检查这个magic cookie，比对是否相同。

举个例子，我们在Windows 7 x64的dhcpcore.dll文件中搜索这个常量。找到两处：看上去这个常量在名为DhcpExtractOptionsForValidation()和DhcpExtractFullOptions()函数中使用:

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h ; DATA XR
EF: ⤦
    DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBEC dword_7                          DATA XR
EF: ⤦
    DhcpExtractFullOptions+97
```

下面是常量被引用的地址：

```
.text:000007FF6480875F  mov    eax, [rsi]
.text:000007FF64808761  cmp    eax, cs:dword_7FF6483CBE8
.text:000007FF64808767  jnz    loc_7FF64817179
```

还有：

```
.text:000007FF648082C7  mov    eax, [r12]
.text:000007FF648082CB  cmp    eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1  jnz    loc_7FF648173AF
```

59.2 搜索常量

在IDA中很容易：使用ALT-B或者ALT-I。如果是在大量文件或者在不可执行文件中搜索常量，我会使用自己编写一个叫binary grep的小工具。

第60章

寻找合适的指令

如果程序使用了FPU指令但使用不多，你可以尝试用调试器手工逐个检查。

举个例子，我们可能会对用户如何在微软的Excel中输入计算公式感兴趣，比如除法操作。

如果我们加载excel.exe(Offic 2010)版本为14.0.4756.1000 到IDA中，列出所有的条目，查找每一条FDIV指令(除了使用常量作为第二个操作数的——显然不是我们所关心的)：

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

然后我们就会看到有144条相关结果。

我们可以在Excel中输入像"=(1/3)"这样的字符串然后对指令进行检查。

通过使用调试器或者tracer(一次性检查4条指令)检查指令，我们幸运地发现目标指令是第14个：

```
.text:3011E919 DC 33        fdiv    qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=
FPU ST(0): 1.000000
```

ST(0)存放了第一个参数，[EBX]存放了第二个参数。

FDIV(FSTP)之后的指令在内存中写入了结果：

```
.text:3011E91B DD 1E        fstp    qword ptr [esi]
```

如果我们设置一个断点，就可以看到结果：

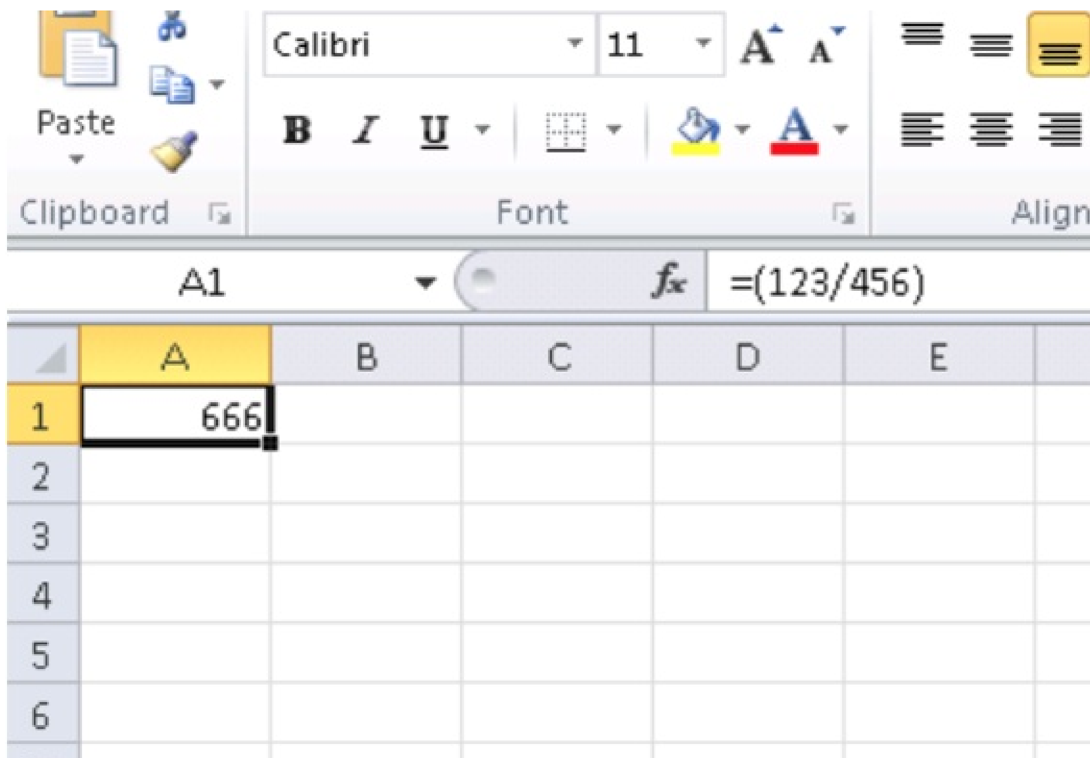
```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

我们也可以恶作剧地修改一下这个值：

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel在这个单元中显示666，我们也可以确信的确找到了正确的位置。



如果我们尝试使用同样的Excel版本，但是是64位的，会发现只有12个FDIV指令，我们的目标指令在第三个。

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

看起来似乎许多浮点数和双精度类型的除法操作都被编译器用SSE指令比如DIVSD(DIVSD总共出现了268次)替换了。

第61章

可疑的代码模式

61.1 XOR 指令

像XOR op这样的指令，op为寄存器(比如，xor eax，eax)通常用于将寄存器的值设置为零，但如果操作数不同，"互斥或"运算将被执行。在普通的程序中这种操作较罕见，但在密码学中应用较广，包括业余的。如果第二个操作数是一个很大的数字，那么就更可疑了。可能会指向加密/解密操作或校验和的计算等等。

而这种观察也可能是无意义的，比如"canary"(18.3节)。canary的产生和检测通常使用XOR指令。

下面这个awk脚本可用于处理IDA的.list文件：

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$
4) if($4!="esp") if ($4!="ebp")
 {print$1,$2,tmp,",",$4}}'filename.lst
```

61.2 Hand-written assembly code

现代编译器不会emit LOOP和RCL指令。另一方面，这些指令对于直接用汇编语言编程的程序员来说很熟悉。如果你发现了这些指令，可以猜测这部分代码极有可能是手工编写的。这样的代码在这个指令列表中用(M)标记：A.6节。

同时函数prologue/epilogue通常不会以手工编写的汇编的形式呈现。

通常情况下，手工编写的代码中参数传递给函数没有固定的系统。

Windows 2003 内核(ntoskrnl.exe 文件)的例子：

```
MultiplyTest    proc near            ; CODE XREF: Get386Stepping
                xor     cx, cx
loc_620555:                            ; CODE XREF: MultiplyTest
+E
                push     cx
                call     Multiply
                pop     cx
                jb         short locret_620563
                loop     loc_620555
                clc
locret_620563:                        ; CODE XREF:MultiplyTest+C
                retn
MultiplyTest endp

Multiply         proc near            ;CODE XREF:MultiplyTest+5
                mov ecx,81h
                mov eax,417A000h
                mul ecx
                cmp edx,2
                stc
                jnz short locret_62057F
                cmp  eax,0FE7A000h
                stc
                jnz short locret_62057F
                clc
locret_62057F:                        ; CODE XREF:Multiply+10
                                     ; Multiply+18
                retn
Multiply        endp
```

事实上，如果我们查看WRK v1.2源码，上面的代码在WRK-v1.2\base\ntos\ke\i386\cpu.asm文件中很容易找到。

第62章

跟踪时使用幻数

通常情况下，我们的主要目标是理解程序从文件读取或从网络中接收的值的用途。手动跟踪某个值常常是个体力活。最简单应对技术之一(尽管不是百分之百靠谱)是使用自定义的幻数。

这在某种程度上类似于X射线计算机断层扫描：造影剂注射到病人的血液中，增强患者的内部结构在X射线下的能见度。例如，健康人的血液在肾脏渗透是众所周知的，如果血液中有介质则可以很容易在断层中看到血液如何渗透的，是否有结石或肿瘤。

我们可以使用一个32比特的数字，比如0x0badf00d，或者某人的生日0x11101979并将这个4字节数字写到我们目标程序使用的文件的某个位置。

然后使用code coverage模式下的tracer的跟踪这个程序，再用grep工具或仅仅是文本搜索(跟踪结果)，就可以轻松看到值的位置以及如何被使用。

使用cc模式下tracer的结果，可使用grep：

```
0x150bf66 (_kziaia+0x14), e=         1 [MOV EBX, [EBP+8]] [EBP+8]
=0xf59c934
0x150bf69 (_kziaia+0x17), e=         1 [MOV EDX, [69AEB08h]] [69A
EB08h]=0
0x150bf6f (_kziaia+0x1d), e=         1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=         1 [MOV ECX, [EAX+EDX*4]] [EA
X+EDX*4]=0xf1ac360
0x150bf78 (_kziaia+0x26), e=         1 [MOV [EBP-4], ECX] ECX=0xf
1ac360
```

对于网络包中也同样适用。很重要的一点是，幻数必须独特保证没有在该程序中出现过。

除了tracer，heavydebug模式下的DosBox(MS-DOS仿真器)也能将每条指令执行后寄存器状态写入到一个文本文件中，因此，这种技术对于DOS程序也是很有用的。

第63章

其他

63.1 基本思想

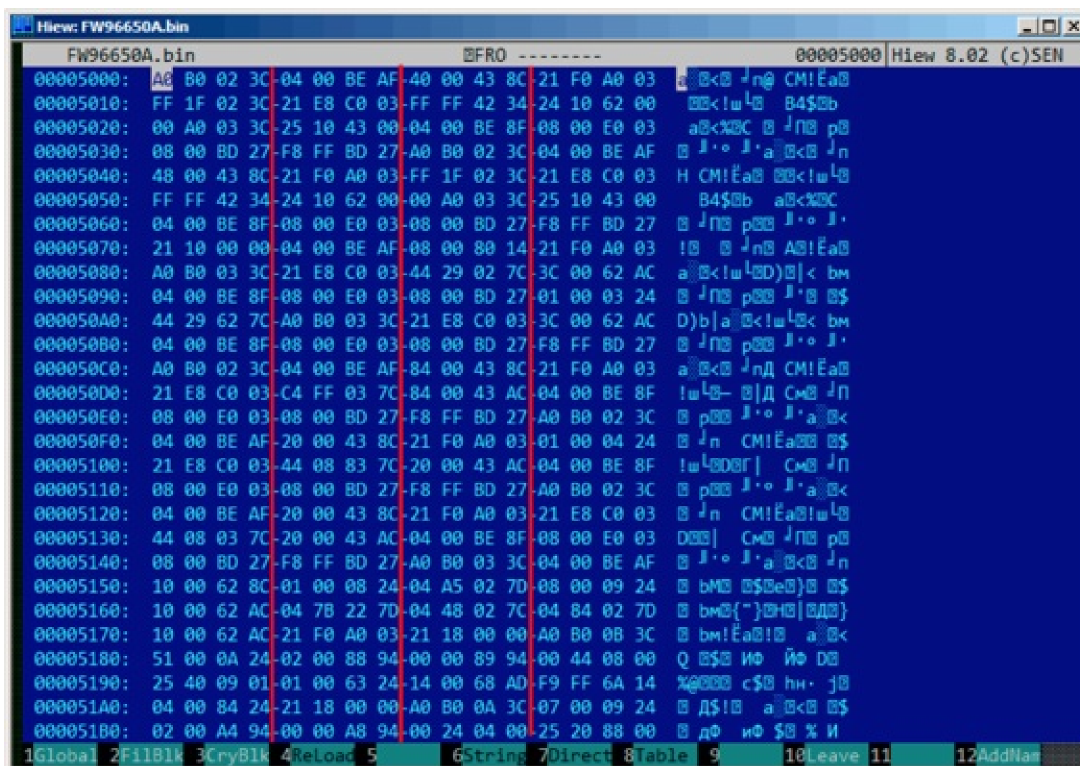
一个逆向工程师应该尽可能多地去尝试站在程序开发者的角度，并思考开发者碰见某些特殊情况会如何解决。

63.2 C++

RTTI(51.1.5)的数据对于C++类定义可能会有帮助。

63.3 某些二进制文件模式

有时我们可以在十六进制编辑器中清楚地看到16/32/64比特值的数组。下面是一个非常典型的MIPS代码。每一个MIPS(还有ARM或ARM64模式的ARM)指令都是32比特(4字节)，构成32比特值的数组。通过查看快照可以看到这种模式。为了显示更清晰我加了红色的下划线：



另一个这种模式的例子：第86节

63.4 内存快照比对

将两个内存快照直接比对来查看变化的技术常用于做8比特的PC游戏的高分游戏挂。

举个例子，如果你在8比特的电脑上加载了一个游戏(这里的内存不多，但游戏需要的内存通常更少)，假设你知道你现在有100发子弹，你可以给内存做个快照放到某处。然后打一发，子弹数变为99，然后再做一个快照进行比对：某处一定会有一个字节一开始是100，现在变成了99。考虑到这些8比特的游戏通常用汇编语言编写，并且这样的变量通常是全局变量，可以确定内存中确有某个地址包含了子弹数目。如果你在反汇编后的游戏代码中搜索了所有有关这个地址的引用，那么找到减少子弹数的代码并不难，然后使用NOP指令替换掉，这样在游戏里子弹数就会一直保持100。通常情况下8比特PC游戏加载地址不变，并且每个游戏的不同版本不多(通常一个版本就会流行很长一段时间)，所以游戏爱好者常常知道哪些地址的哪些字节需要覆盖(使用BASIC指令POKE)。由此形成了一个包含了POKE指令游戏挂，发布在和8比特游戏有关的杂志上。见:wikipedia

同样的，修改高分文件也很容易，并且不仅仅是处理8比特游戏了。记下你的得分数并且将文件备份。当高分变化后将两个文件进行比对，使用DOS的FC工具就可以(高分文件通常是二进制形式)。某处一定会有部分字节不同，发现哪些字节包含了得分数很容易。然而，游戏开发者为了防范这些游戏挂可能会采取一些措施。

这本书中其他类似的例子：第85节

63.4.1 Windows注册表

在程序安装前后比对注册表的变化也是可行的，常用于寻找与程序有关的注册表元素。这也可能是"windows registry cleaner"共享软件如此受欢迎的原因吧。

63.4.2 Blink-comparator

文件或内存快照的比对让我们想起了blink-comparator：一种曾被天文学家使用的设备，用于发现天体移动。blink-comparator允许在两个不同时间摄影快照间切换，便于天文学家发现差别。顺便说一句，冥王星就是在1930年用blink-comparator发现的。

Part VI 操作系统的特性

第六十四章

传递参数的方法

64.1 cdcel

这种传递参数的方法在C/C++语言里面比较流行。

如下的代码片段所示，调用者反序地把参数压到栈中：最后一个参数，倒数第二个参数，第一个参数。调用者还必须在函数返回之后把栈指针（ESP）还原为初始状态。

Listing 64.1: cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

64.2 stdcall

该调用方法与cdecl差不多，除了被调用者必须通过RET x指令代替RET指令将ESP指针设置为初始化状态，其中 `x = arguments number * sizeof(int)` 。调用者无需调整栈指针(ESP)。

Listing 64.2: stdcall

```
push arg3
push arg2
push arg1
call function
function:
... do something ...
ret 12
```

这种调用方式在win32的标准库无处不在，但win64并不使用该调用方法（具体参见下文win64一节）。

举个例子，我们可以稍微把在91页中8.1的示例代码修改一下，增加一个 `__stdcall` 修饰符。

```
int __stdcall f2 (int a, int b, int c)
{
    return a*b+c;
};
```

编译出来的结果跟8.2几乎一模一样，但你可以看到它是通过RET 12而不是RET返回的。同时，调用者并没有调整栈指针(ESP)。

因此，很容易通过RETN n指令推导出函数参数的数量（n除以四）。

Listing 64.3: MSVC 2010

```
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_c$ = 16 ; size = 4
_f2@12 PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    imul eax, DWORD PTR _b$[ebp]
    add eax, DWORD PTR _c$[ebp]
    pop ebp
    ret 12 ; 0000000cH
_f2@12 ENDP
; ...
    push 3
    push 2
    push 1
    call _f2@12
    push eax
    push OFFSET $SG81369
    call _printf
    add esp, 8
```

64.2.1 可变参数的函数

printf()系列的函数大概是C/C++里面唯一一系列具有可变参数的函数了，在这些函数的帮助下很容易理清cdecl和stdcall两种调用方式之间的重要区别。让我们先假设编译器知道每个调用printf()函数的参数的个数，无论如何，当我们调用printf()的时候，它已经存在于编译好的MSVCRT.DLL之中（我们讨论的是Windows），并没有任何关于传递多少个参数的信息，剩下的办法就是通过它的格式字符串获取得到参数个数。因此，如果printf()函数是一个stdcall调用方式的函数，它必须通过格式字符串计算参数个数用于恢复栈指针（ESP），这是一种相当危险的情况，程序员的一个错别字就可以导致程序崩溃。因此此类函数使用cdecl调用方式远比使用stdcall调用方式适合。

64.3 fastcall

这是一种将部分参数通过寄存器传入，其余参数通过栈方式传入的方法。它的执行效率在一些旧时CPU比cdecl/stdcall要好（因为小栈的压力）。然而，在现代的CPU中使用该调用方式不一定能获得更好的性能。

fastcall并没有一个标准化，因此不同的编译器的实现可以不同。这是一个众所周知的警告：如果你有两个DLL，其中第一个DLL调用第二个DLL的函数，它们是又分别不同的编译器使用fastcall调用方式编译出来的，则会有不可预期的后果。

MSVC和GCC两个编译器都是通过ECX和EDX来传递第一个和第二个参数，通过栈进行传递其余参数。栈指针必须被被调用者恢复为初始状态（与stdcall类似）。

Listing 64.4: fastcall

```
push arg3
mov edx, arg2
mov ecx, arg1
call function
function:
.. do something ..
ret 4
```

举个例子，我们可以稍微把8.1的示例代码修改一下，增加一个 `__fastcall` 修饰符。

```
int __fastcall f3 (int a, int b, int c)
{
    return a*b+c;
};
```

下面它编译出来的结果：

Listing 64.5: Optimizing MSVC 2010 /Ob0

```
_c$ = 8         ; size = 4
@f3@12 PROC
; _a$ = ecx
; _b$ = edx
    mov eax, ecx
    imul eax, edx
    add eax, DWORD PTR _c$[esp-4]
    ret 4
@f3@12 ENDP
; ...
    mov edx, 2
    push 3
    lea ecx, DWORD PTR [edx-1]
    call @f3@12
    push eax
    push OFFSET $SG81390
    call _printf
    add esp, 8
```

我们可以看到被调用者使用RET N指令来调整栈指针（ESP）。这意味着，我们可以通过这条指令来推断出参数的个数。

64.3.1 GCC regparm

这是一种对fastcall调用方式的某种优化。使用-mregparm编译选项可以设置多少个参数是通过寄存器传递的（最大为3个）。因此，EAX，EDX和ECX寄存器将被使用。

当然，如果指定通过寄存器传参的参数数量小于三个的时候，并没有使用完这三个寄存器。

调用者需要把栈指针恢复为初始状态。

相关例子请参看(19.1.1)。

64.3.2 Watcom/OpenWatcom 编译器

在这里，它被成为“寄存器调用约定”，头四个参数通过EAX，EDX，EBX和ECX传递。其余参数通过栈传递。通过在函数名上添加下划线来区分那些不同的调用约定。

64.4 thiscall

这是C++里面传递this指针的成员函数调用约定。

在MSVC里面，this指针通过ECX寄存器来传递。

在GCC里面，this指针是通过第一个参数进行传递的。因此很明显，在所有成员函数里面都会多出一个额外的参数。

相关例子请查看（51.1.1）。

64.5 x86-64

64.5.1 Windows x64

在Win64里面传递函数参数的方法类似fastcall调用约定。前四个参数通过RCX，RDX，R8和R9寄存器传参，其余参数通过栈进行传递。调用者还必须预留32个字节或者4个64位的空间，让被调用者可以保存前四个参数。短函数可能直接使用通过寄存器传过来的值，但更大的可能是保存那些值后在进一步使用。

调用者还必须负责还原栈指针。

这个调用约定也用于Windows x86-64位系统上的DLL（而不是Win32的stdcall）。

例子

```
#include <stdio.h>
void f1(int a, int b, int c, int d, int e, int f, int g)
{
    printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};
int main()
{
    f1(1,2,3,4,5,6,7);
};
```

Listing 64.6: MSVC 2012 /0b

```
$SG2937 DB '%d %d %d %d %d %d %d', 0aH, 00H
main PROC
    sub rsp, 72                      ; 00000048H
    mov DWORD PTR [rsp+48], 7
    mov DWORD PTR [rsp+40], 6
    mov DWORD PTR [rsp+32], 5
    mov r9d, 4
    mov r8d, 3
    mov edx, 2
    mov ecx, 1
    call f1
    xor eax, eax
    add rsp, 72                      ; 00000048H
    ret 0
main ENDP
a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1 PROC
$LN3:
    mov DWORD PTR [rsp+32], r9d
    mov DWORD PTR [rsp+24], r8d
    mov DWORD PTR [rsp+16], edx
    mov DWORD PTR [rsp+8], ecx
    sub rsp, 72                      ; 00000048H
    mov eax, DWORD PTR g$[rsp]
    mov DWORD PTR [rsp+56], eax
    mov eax, DWORD PTR f$[rsp]
    mov DWORD PTR [rsp+48], eax
    mov eax, DWORD PTR e$[rsp]
    mov DWORD PTR [rsp+40], eax
    mov eax, DWORD PTR d$[rsp]
    mov DWORD PTR [rsp+32], eax
    mov r9d, DWORD PTR c$[rsp]
    mov r8d, DWORD PTR b$[rsp]
    mov edx, DWORD PTR a$[rsp]
    lea rcx, OFFSET FLAT:$SG2937
    call printf
    add rsp, 72                      ; 00000048H
    ret 0
    f1 ENDP
```

在这里我们可以清楚看到这7个参数是如何传递的：4个参数通过寄存器传递而其余3个通过栈传递。f1()的反汇编代码一开始就把参数保存到“预留”的栈空间之中，这样做的目的是编译器并不能保证有足够的寄存器可以使用，如果不这样做的话这四个寄存器将被参数占用到函数执行结束。最后，预留栈空间是调用者的职责。

Listing 64.7: Optimizing MSVC 2012 /0b

```
$SG2777 DB '%d %d %d %d %d %d %d', 0aH, 00H
a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1 PROC
$LN3:
    sub rsp, 72                         ; 00000048H
    mov eax, DWORD PTR g$[rsp]
    mov DWORD PTR [rsp+56], eax
    mov eax, DWORD PTR f$[rsp]
    mov DWORD PTR [rsp+48], eax
    mov eax, DWORD PTR e$[rsp]
    mov DWORD PTR [rsp+40], eax
    mov DWORD PTR [rsp+32], r9d
    mov r9d, r8d
    mov r8d, edx
    mov edx, ecx
    lea rcx, OFFSET FLAT:$SG2777
    call printf
    add rsp, 72                         ; 00000048H
    ret 0
f1 ENDP
main PROC
    sub rsp, 72                         ; 00000048H
    mov edx, 2
    mov DWORD PTR [rsp+48], 7
    mov DWORD PTR [rsp+40], 6
    lea r9d, QWORD PTR [rdx+2]
    lea r8d, QWORD PTR [rdx+1]
    lea ecx, QWORD PTR [rdx-1]
    mov DWORD PTR [rsp+32], 5
    call f1
    xor eax, eax
    add rsp, 72                         ; 00000048H
    ret 0
main ENDP
```

如果我们使用了编译优化的开关去编译上面的例子，它的反汇编码几乎是相同的，但是预留的栈空间将不被使用，因为在这里并不需要使用到预留的栈空间。

而且可以看到MSVC 2012是如何利用LEA指令来优化代码（A.6.2）。

我也不确定是否值得这么做。

更多的例子请看（74.1）

this指针的传递(C/C++)

this指针通过RCX传递，成员函数的第一个参数通过RDX传递，更多例子请看（51.1.1）。

64.5.2 Linux x64

Linux x86-64传递参数的方式几乎和Windows一样。但是是通过6个寄存器代替4个寄存器来传参（RDI，RSI，RDX，RCX，R8，R9），另外并没有预留的栈空间这回事。虽然，如果它需要/想要的话，可以把寄存器的值保存到栈之中。

Listing 64.8: Optimizing GCC 4.7.3

```
.LC0:
    .string "%d %d %d %d %d %d %d\n"
f1:
    sub rsp, 40
    mov eax, DWORD PTR [rsp+48]
    mov DWORD PTR [rsp+8], r9d
    mov r9d, ecx
    mov DWORD PTR [rsp], r8d
    mov ecx, esi
    mov r8d, edx
    mov esi, OFFSET FLAT:.LC0
    mov edx, edi
    mov edi, 1
    mov DWORD PTR [rsp+16], eax
    xor eax, eax
    call __printf_chk
    add rsp, 40
    ret
main:
    sub rsp, 24
    mov r9d, 6
    mov r8d, 5
    mov DWORD PTR [rsp], 7
    mov ecx, 4
    mov edx, 3
    mov esi, 2
    mov edi, 1
    call f1
    add rsp, 24
    ret
```

注意：这里的值是写入到32-bit的寄存器（EAX...）而不是整个64-bit寄存器（RAX...）。这是因为写入到32-bit寄存器的时候会自动清空高32-bit。据说，这是为了方便把代码移植到x86-64。

64.6 返回float和double类型的值

除了Win64之外，其它返回float和double类型的值都是通过FPU里面的ST(0)寄存器返回的。 在Win64里面，返回float和double类型的值是通过XMM0寄存器返回。

64.7 修改参数

有时候，C/C++程序员（虽然不仅仅是这些人）可能会问，如果他们碰巧修改了参数会怎样？答案非常简单，这些参数是保存在栈里面的，修改参数的时候是修改这个栈里面的内容，调用者并没有在被调用函数退出之后再使用它们（至少在我的实践中没有遇到这种情况）。

```
#include <stdio.h>
void f(int a, int b)
{
    a=a+b;
    printf ("%d\n", a);
};
```

Listing 64.9: MSVC 2012

```
_a$ = 8     ; size = 4
_b$ = 12     ; size = 4
_f PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    add eax, DWORD PTR _b$[ebp]
    mov DWORD PTR _a$[ebp], eax
    mov ecx, DWORD PTR _a$[ebp]
    push ecx
    push OFFSET $SG2938 ; '%d', 0aH
    call _printf
    add esp, 8
    pop ebp
    ret 0
_f END
```

是的，可以随便修改参数。当然，这得它不是C++的引用（references）（51.3），而且你如果不修改通过指针指向的数据。那么修改参数是不会影响到当前函数的。

从理论上来讲，被调用者的函数返回之后，调用者可以获取并修改和使用它。如果它是直接使用汇编语言编写的。但C/C++并不提供任何方式可以访问它们。

64.8 使用指针的函数参数

...更有意思的是，有可能在程序中，取一个函数参数的指针并将其传递给另外一个函数。

```
#include <stdio.h>
// located in some other file
void modify_a (int *a);
void f (int a)
{
    modify_a (&a);
    printf ("%d\n", a);
};
```

很难理解它是如果实现的，直到我们看到它的反汇编码：

Listing 64.10: Optimizing MSVC 2010

```
$SG2796 DB '%d', 0aH, 00H
_a$ = 8
_f PROC
    lea eax, DWORD PTR _a$[esp-4]     ; just get the address of
value in local stack
    push eax                         ; and pass it to modify_a()
    call _modify_a
    mov ecx, DWORD PTR _a$[esp]     ; reload it from the local s
tack
    push ecx                         ; and pass it to printf()
    push OFFSET $SG2796             ; '%d'
    call _printf
    add esp, 12
    ret 0
_f ENDP
```

传递到另一个函数是a在栈空间上的地址，该函数修改了指针指向的值然后再调用printf()来打印出修改之后的值。

细心的读者可能会问，使用寄存器传参的调用约定是如何传递函数指针参数的？

这是一种利用了影子空间的情况，输入的参数值先从寄存器复制到局部栈中的影子空间，然后再讲这个地址传递给其他函数。

Listing 64.11: Optimizing MSVC 2012 x64

```
$SG2994 DB '%d', 0aH, 00H
a$ = 48
f PROC
    mov DWORD PTR [rsp+8], ecx          ; save input value in Sha
dow Space
    sub rsp, 40
    lea rcx, QWORD PTR a$[rsp]          ; get address of value an
d pass it to modify_a()
    call modify_a
    mov edx, DWORD PTR a$[rsp]          ; reload value from Shado
w Space and pass it to printf()
    lea rcx, OFFSET FLAT:$SG2994    ; '%d'
    call printf
    add rsp, 40
    ret 0
f ENDP
```

GCC同样将传入的参数存储在本地栈空间：

Listing 64.12: Optimizing GCC 4.9.1 x64

```
.LC0:
.string "%d\n"
f:
    sub rsp, 24
    mov DWORD PTR [rsp+12], edi     ; store input value to the l
ocal stack
    lea rdi, [rsp+12]                   ; take an address of the v
alue and pass it to modify_a()
    call modify_a
    mov edx, DWORD PTR [rsp+12]     ; reload value from the loca
l stack and pass it to printf()
    mov esi, OFFSET FLAT:.LC0           ; '%d'
    mov edi, 1
    xor eax, eax
    call __printf_chk
    add rsp, 24
    ret
```

ARM64的GCC也做了同样的事情，但这个空间称为寄存器保护区：

```
f:
    stp x29, x30, [sp, -32]!
    add x29, sp, 0         ; setup FP
    add x1, x29, 32     ; calculate address of variable in Regis
ter Save Area
    str w0, [x1,-4]!     ; store input value there
    mov x0, x1             ; pass address of variable to the mod
ify_a()
    bl modify_a
    ldr w1, [x29,28]     ; load value from the variable and pass
 it to printf()
    adrp x0, .LC0 ; '%d'
    add x0, x0, :lo12:.LC0
    bl printf ; call printf()
    ldp x29, x30, [sp], 32
    ret
.LC0:
    .string "%d\n"
```

顺便提一下，一个类似影子空间的使用在这里也被提及过（46.1.2）。

第六十五章

线程局部存储

TLS是每个线程特有的数据区域，每个线程可以把自己需要的数据存储在这里。一个著名的例子是C标准的全局变量errno。多个线程可以同时使用errno获取返回的错误码，如果是全局变量它是无法在多线程环境下正常工作的。因此errno必须保存在TLS。

C++11标准里面新添加了一个thread_local修饰符，标明每个线程都属于自己版本的变量。它可以被初始化并位于TLS中。

Listing 65.1: C++11

```
#include <iostream>
#include <thread>
thread_local int tmp=3;
int main()
{
    std::cout << tmp << std::endl;
};
```

使用MinGW GCC 4.8.1而不是MSVC2012编译。

如果我们查看它的PE文件，可以看到tmp变量被放到TLS section。

65.1 线性同余发生器

前面第20章的纯随机数生成器有一个缺陷：它不是线程安全的，因为它的内部状态变量可以被不同的线程同时读取或修改。

65.1.1 Win32

未初始化的TLS数据

一个全局变量如果添加了_declspec(thread)修饰符，那么它会被分配在TLS。

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    my_srand(0x12345678);
    printf ("%d\n", my_rand());
};
```

使用Hiew可以看到PE文件多了一个section：.tls。

Listing 65.2: Optimizing MSVC 2013 x86

```
_TLS SEGMENT
    _rand_state DD 01H DUP (?)
_TLS ENDS

_DATA SEGMENT
    $SG84851 DB '%d', 0aH, 00H
_DATA ENDS

_TEXT SEGMENT

_init$ = 8  ; size = 4

_my_srand PROC
; FS:0=address of TIB
    mov eax, DWORD PTR fs:__tls_array ; displayed in IDA as FS:2
Ch
; EAX=address of TLS of process
    mov ecx, DWORD PTR __tls_index
    mov ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
    mov eax, DWORD PTR _init$[esp-4]
    mov DWORD PTR _rand_state[ecx], eax
    ret 0
_my_srand ENDP

_my_rand PROC
; FS:0=address of TIB
    mov eax, DWORD PTR fs:__tls_array ; displayed in IDA as FS:2
Ch
; EAX=address of TLS of process
    mov ecx, DWORD PTR __tls_index
    mov ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
    imul eax, DWORD PTR _rand_state[ecx], 1664525
    add eax, 1013904223 ; 3c6ef35fH
    mov DWORD PTR _rand_state[ecx], eax
    and eax, 32767 ; 00007fffH
    ret 0
_my_rand ENDP

_TEXT ENDS
```

rand_state现在处于TLS段，而且这个变量每个线程都拥有属于自己版本。它是这么访问的：从FS:2Ch加载TIB（Thread Information Block）的地址，然后添加一个额外的索引（如果需要的话），接着计算出在TLS段的地址。

最后可以通过ECX寄存器来访问rand_state变量，它指向每个线程特定的数据区域。

FS：这是每个逆向工程师都很熟悉的选择子了。它专门用于指向TIB，因此访问线程特定数据可以很快完成。

GS: 该选择子用于Win64，0x58的地址是TLS。

Listing 65.3: Optimizing MSVC 2013 x64

```
_TLS SEGMENT
    rand_state DD 01H DUP (?)
_TLS ENDS

_DATA SEGMENT
    $SG85451 DB '%d', 0aH, 00H
_DATA ENDS

_TEXT SEGMENT
init$ = 8

my_srand PROC
    mov edx, DWORD PTR _tls_index
    mov rax, QWORD PTR gs:88 ; 58h
    mov r8d, OFFSET FLAT:rand_state
    mov rax, QWORD PTR [rax+rdx*8]
    mov DWORD PTR [r8+rax], ecx
    ret 0
my_srand ENDP

my_rand PROC
    mov rax, QWORD PTR gs:88 ; 58h
    mov ecx, DWORD PTR _tls_index
    mov edx, OFFSET FLAT:rand_state
    mov rcx, QWORD PTR [rax+rcx*8]
    imul eax, DWORD PTR [rcx+rdx], 1664525 ;0019660dH
    add eax, 1013904223 ; 3c6ef35fH
    mov DWORD PTR [rcx+rdx], eax
    and eax, 32767 ; 00007fffH
    ret 0
my_rand ENDP

_TEXT ENDS
```

初始化TLS数据

比方说，我们想为rand_state设置一些固定的值以避免程序员忘记初始化。

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state=1234;

void my_srand (uint32_t init)
{
   rand_state=init;
}

int my_rand ()
{
   rand_state=rand_state*RNG_a;
   rand_state=rand_state+RNG_c;
   return rand_state & 0x7fff;
}

int main()
{
    printf ("%d\n", my_rand());
};
```

代码除了给rand_state设定初始值外与之前的并没有什么不同，但在IDA我们看到：

```
.tls:00404000 ; Segment type: Pure data
.tls:00404000 ; Segment permissions: Read/Write
.tls:00404000 _tls segment para public 'DATA' use32
.tls:00404000 assume cs:_tls
.tls:00404000 ;org 404000h
.tls:00404000 TlsStart db 0 ; DATA XREF: .rdata:TlsDirectory
.tls:00404001 db 0
.tls:00404002 db 0
.tls:00404003 db 0
.tls:00404004 dd 1234
.tls:00404008 TlsEnd db 0 ; DATA XREF: .rdata:TlsEnd_pt
...
```

每次一个新的线程运行的时候，会分配新的TLS给它，然后包括1234所有数据将被拷贝过去。

这是一个典型的场景：

- 线程A开始运行，然后分配给它一个TLS，并把1234拷贝到rand_state。
- 线程A里面多次调用my_rand()函数，rand_state已经不是1234。

- 线程B开始运行，然后分配给它一个TLS，并把1234拷贝到rand_state，这时候可以观察到两个线程使用同一个变量，但它们的值是不一样的。

TLS callbacks

如果我们想给TLS赋一个变量值呢？比方说：程序员忘记调用my_srand()函数来初始化PRNG，但是随机数生成器在开始的时候必须使用一个真正的随机数值而不是1234。这种情况下则可以使用TLS callbaks。

下面的代码的可移植性很差，原因你应该明白。我们定义了一个函数(tls_callback())，它在进程/线程开始执行前调用，该函数使用GetTickCount()函数的返回值来初始化PRNG。

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
    my_srand (GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}
int main()
{
    // rand_state is already initialized at the moment (using Ge
tTickCount())
    printf ("%d\n", my_rand());
};
```

用IDA看一下：

Listing 65.4: Optimizing MSVC 2013

```
.text:00401020 TlsCallback_0 proc near ; DATA XREF: .rdata:TlsCa
llbacks
.text:00401020     call ds:GetTickCount
.text:00401026     push eax
.text:00401027     call my_srand
.text:0040102C     pop ecx
.text:0040102D     retn 0Ch
.text:0040102D TlsCallback_0 endp
...
.rdata:004020C0 TlsCallbacks dd offset TlsCallback_0 ; DATA XREF
: .rdata:TlsCallbacks_ptr
...
.rdata:00402118 TlsDirectory dd offset TlsStart
.rdata:0040211C TlsEnd_ptr dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h
```

TLS callbacks函数时常用于隐藏解包处理过程。为此有些人可能会困惑，为什么一些代码可以偷偷地在OEP（Original Entry Point）之前执行。

65.1.2 Linux

下面是GCC声明线程局部存储的方式：

```
__thread uint32_t rand_state=1234;
```

这不是标准C/C++的修饰符，但是是GCC的一个扩展特性。

GS：该选择子同样用于访问TLS，但稍微有点区别：

Listing 65.5: Optimizing GCC 4.8.1 x86

```
.text:08048460 my_srand proc near
.text:08048460
.text:08048460 arg_0 = dword ptr 4
.text:08048460
.text:08048460     mov eax, [esp+arg_0]
.text:08048464     mov gs:0FFFFFFFCh, eax
.text:0804846A     retn
.text:0804846A my_srand endp
.text:08048470 my_rand proc near
.text:08048470     imul eax, gs:0FFFFFFFCh, 19660Dh
.text:0804847B     add eax, 3C6EF35Fh
.text:08048480     mov gs:0FFFFFFFCh, eax
.text:08048486     and eax, 7FFFh
.text:0804848B     retn
.text:0804848B my_rand endp
```

更多例子：ELF Handling For Thread-Local Storage

第六十六章

系统调用(syscall-s)

众所周知，所有运行的进程在操作系统里面分为两类：一类拥有访问全部硬件设备的权限（内核空间）而另一类无法直接访问硬件设备(用户空间)。

操作系统内核和驱动程序通常是属于第一类的。

而应用程序通常是属于第二类的。

举个例子，Linux kernel运行于内核空间，而Glibc运行于用户空间。

这种分离对与操作系统的安全性是至关重要的：它最重要的一点是，不给任何进程有破坏到其它进程甚至是系统内核的机会。另一方面，一个错误的驱动或系统内核错误都会造成系统崩溃或者蓝屏。

保护模式下的x86处理器允许使用4个保护等级（ring）。但Linux和Windows两个操作系统都只使用了两个：ring0（内核空间）和ring3（用户空间）。

系统调用（syscall-s）是两个运行空间的连接点。可以说，这是提供给应用程序主要的API。

在Windows NT，系统调用表存在于SSDT。

通过系统调用实现shellcode在计算机病毒作者之间非常流行。因为很难确定所需函数在系统库里面的地址，但系统调用很容易确定。然而，由于系统调用属于比较底层的API，所以需要编写更多的代码。最后值得一提的是，在不同的操作系统版本里面，系统调用号是有可能不同的。

66.1 Linux

在Linux系统中，系统调用通常使用int 0x80中断进行调用。通过EAX寄存器传递调用号，再通过其它寄存器传递所需参数。

Listing 66.1: A simple example of the usage of two syscalls

```
section .text
global _start
_start:
    mov edx,len    ; buf len
    mov ecx,msg    ; buf
    mov ebx,1    ; file descriptor. stdout is 1
    mov eax,4    ; syscall number. sys_write is 4
    int 0x80
    mov eax,1    ; syscall number. sys_exit is 4
    int 0x80
section .data
msg db 'Hello, world!',0xa
len equ $ - msg
```

编译：

```
nasm -f elf32 1.s
ld 1.o
```

Linux所有的系统调用在这里可以查看：http://go.yurichev.com/17319。

在Linux中可以使用strace(71章)对系统调用进行跟踪或者拦截。

66.2 Windows

Windows系统使用int 0x2e中断或x86下特有的指令SYSENTER调用用系统调用服务。

Windows所有的系统调用在这里可以查看：http://go.yurichev.com/17320。

扩展阅读："Windows Syscall Shellcode" by Piotr Bania

第六十七章

Linux

67.1 位置无关代码

在分析Linux共享库(.so)的时候，可能会经常看到类似下面的代码：

Listing 67.1: libc-2.17.so x86

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near ; CODE XREF: sub_
17350+3
.text:0012D5E3 ; sub_173CC+4 ...
.text:0012D5E3     mov ebx, [esp+0]
.text:0012D5E6     retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp
...
.text:000576C0 sub_576C0 proc near ; CODE XREF: tmpfile+73
...
.text:000576C0     push ebp
.text:000576C1     mov ecx, large gs:0
.text:000576C8     push edi
.text:000576C9     push esi
.text:000576CA     push ebx
.text:000576CB     call __x86_get_pc_thunk_bx
.text:000576D0     add ebx, 157930h
.text:000576D6     sub esp, 9Ch
...
.text:000579F0     lea eax, (a__gen_tempname - 1AF000h)[ebx] ; "
__gen_tempname"
.text:000579F6     mov [esp+0ACh+var_A0], eax
.text:000579FA     lea eax, (a__SysdepsPosix - 1AF000h)[ebx] ; "
../sysdeps/posix/tempname.c"
.text:00057A00     mov [esp+0ACh+var_A8], eax
.text:00057A04     lea eax, (aInvalidKindIn_ - 1AF000h)[ebx] ; "
! \"invalid KIND in __gen_tempname\""
.text:00057A0A     mov [esp+0ACh+var_A4], 14Ah
.text:00057A12     mov [esp+0ACh+var_AC], eax
.text:00057A15     call __assert_fail
```

所有指向字符串的指针都需要通过在每个函数开始处计算的EBX值和一些常量值来修正地址。这就是所谓的PIC（位置无关代码），它的目的是让这段代码即使放在内存中任何随机位置都能正确地执行。这也是为什么不能使用绝对地址的原因。

PIC（位置无关代码）对于早期的操作系统和现在那些没有虚拟内存支持的嵌入式系统来说至关重要（所有进程都放在同一个连续的内存块）。此外，它还用于*NIX系统的共享库。这样共享库只需要加载一次到内存之后就可以让所有需要的进程使用。而且这些进程可以把同一个共享库映射到各自不同的内存地址上。这也是为什么共享库不使用绝对地址也能够正常地工作的原因。

让我们做一个简单的实验：

```
#include <stdio.h>
int global_variable=123;
int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

用GCC 4.7.3编译它并用IDA查看.so文件的反汇编代码：

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

```
.text:00000440 public __x86_get_pc_thunk_bx
.text:00000440 __x86_get_pc_thunk_bx proc near ; CODE XREF: _ini
t_proc+4
.text:00000440 ; deregister_tm_clones+4 ...
.text:00000440     mov ebx, [esp+0]
.text:00000443     retn
.text:00000443 __x86_get_pc_thunk_bx endp
.text:00000570 public f1
.text:00000570 f1 proc near
.text:00000570
.text:00000570 var_1C = dword ptr -1Ch
.text:00000570 var_18 = dword ptr -18h
.text:00000570 var_14 = dword ptr -14h
.text:00000570 var_8 = dword ptr -8
.text:00000570 var_4 = dword ptr -4
.text:00000570 arg_0 = dword ptr 4
.text:00000570
.text:00000570     sub esp, 1Ch
.text:00000573     mov [esp+1Ch+var_8], ebx
.text:00000577     call __x86_get_pc_thunk_bx
.text:0000057C     add ebx, 1A84h
.text:00000582     mov [esp+1Ch+var_4], esi
.text:00000586     mov eax, ds:(global_variable_ptr - 2000h)[ebx
]
.text:0000058C     mov esi, [eax]
.text:0000058E     lea eax, (aReturningD - 2000h)[ebx] ; "return
ing %d\n"
.text:00000594     add esi, [esp+1Ch+arg_0]
.text:00000598     mov [esp+1Ch+var_18], eax
.text:0000059C     mov [esp+1Ch+var_1C], 1
.text:000005A3     mov [esp+1Ch+var_14], esi
.text:000005A7     call ___printf_chk
.text:000005AC     mov eax, esi
.text:000005AE     mov ebx, [esp+1Ch+var_8]
.text:000005B2     mov esi, [esp+1Ch+var_4]
.text:000005B6     add esp, 1Ch
.text:000005B9     retn
.text:000005B9 f1 endp
```

如上所示：每个函数执行时都会修正指向“returning %d\n”和global_variable的指针。__x86_get_pc_thunk_bx()函数自身调用后在EBX返回一个指针（这里是0x57C）。这是一种获取程序计数器（EIP）的简单方法。0x1A84常量是这个函数开始处到 Global Offset Table Procedure Linkage Table(GOT PLT) 它们之间的距离差。IDA会把这些偏移处理成更容易理解后再显示出来，所以实际上的代码是：

```
.text:00000577 call __x86_get_pc_thunk_bx
.text:0000057C add ebx, 1A84h
.text:00000582 mov [esp+1Ch+var_4], esi
.text:00000586 mov eax, [ebx-0Ch]
.text:0000058C mov esi, [eax]
.text:0000058E lea eax, [ebx-1A30h]
```

这里的EBX指向了GOT PLT section。当计算global_variable（存储在GOT）的地址时须减去0x0C偏移量。当计算"returning %d\n"字符串的地址时须减去0x1A30偏移量。

顺便说一下，AMD64的指令支持使用RIP用于相对寻址，这使得它可以产生出更简洁的PIC代码。

让我们用相同的GCC编译器编译相同的C代码，但使用x64平台。

IDA会简化了反汇编代码，造成我们无法看到使用RIP相对寻址的细节，所以我在这里使用了objdump来查看反汇编代码：

```
0000000000000720 <f1>:
720: 48 8b 05 b9 08 20 00    mov rax,QWORD PTR [rip+0x2008b9] #
200fe0 <_DYNAMIC+0x1d0>
727: 53                      push rbx
728: 89                      fb mov ebx,edi
72a: 48 8d 35 20 00 00 00    lea rsi,[rip+0x20] #751 <_fini+0x9>
731: bf 01 00 00 00          mov edi,0x1
736: 03 18                   add ebx,DWORD PTR [rax]
738: 31 c0                   xor eax,eax
73a: 89 da                   mov edx,ebx
73c: e8 df fe ff ff          call 620 <__printf_chk@plt>
741: 89 d8                   mov eax,ebx
743: 5b                      pop rbx
744: c3                      ret
```

0x2008b9是0x720处指令地址到global_variable地址的差，0x20是0x72a处指令地址到"returning %d\n"字符串地址的差。

你可能会看到，频繁重新计算地址会导致执行效率变差（虽然在x64会更好）。所以如果你比较关心性能的话最好还是使用静态链接。

67.1.1 Windows

Windows的DLL并没有使用PIC机制。如果Windows加载器需加载DLL到另外一个基地址，它会在内存中（在固定的位置）对DLL "打补丁" 来将所有地址都调整为正确的。这意味着多个Windows进程不能在不同进程内存块的不同地址共享一份DLL，因为每个实例加载在内存后只固定在这些地址工作。

67.2 *LD_PRELOAD* hack in Linux

Linux允许让我们自己的动态链接库加载在其它动态链接库之前，甚至是系统库（如libc.so.6）。

反过来想，也就是允许我们用自己写的函数去“代替”系统库的函数。举个例子，我们可以很容易地拦截掉time()，read()，write()等等这些函数。

来瞧瞧我们是如何愚弄uptime这个程序的。我们知道，该程序显示计算机已经工作了多长时间。借助strace的帮助可以看到，该程序通过/proc/uptime文件获取到计算机的工作时长。

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY) = 3
lseek(3, 0, SEEK_SET) = 0
read(3, "416166.86 414629.38\n", 2047) = 20
...
```

/proc/uptime并不是存放在磁盘的眞实文件。而是由Linux Kernel 产生一个虚拟的文件。它有两个数值：

```
$ cat /proc/uptime
416690.91 415152.03
```

我们可以用wikipedia来看一下它的含义：

```
第一个数值是系统运行总时长，第二个数值是系统空闲的时间。都以秒为单位表示。
```

我们来写一个含open()，read()，close()函数的动态链接库。

首先，我们的open()函数会比较一下文件名是不是我们所想要打开的，如果是，则将文件描述符记录下来。然后，read()函数会判断如果我们调用的是不是我们所保存的文件描述符，如果是则代替它输出，否则调用libc.so.6里面原来的函数。最后，close()函数会关闭我们所保存的文件描述符。

在这里我们借助了dlopen()和dlsym()函数来确定原先在libc.so.6的函数的地址，因为我们需要控制“眞实”的函数。

题外话，如果我们的程序想劫持strcmp()函数来监控每个字符串的比较，则需要我们自己实现一个strcmp()函数而不能用原先的函数。

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
```

```
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;
bool inited = false;

_Noreturn void die (const char * fmt, ...)
{
    va_list va;
    va_start (va, fmt);
    vprintf (fmt, va);
    exit(0);
};

static void find_original_functions ()
{
    if (inited)
        return;
    libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
    if (libc_handle==NULL)
        die ("can't open libc.so.6\n");
    open_ptr = dlsym (libc_handle, "open");
    if (open_ptr==NULL)
        die ("can't find open()\n");
    close_ptr = dlsym (libc_handle, "close");
    if (close_ptr==NULL)
        die ("can't find close()\n");
    read_ptr = dlsym (libc_handle, "read");
    if (read_ptr==NULL)
        die ("can't find read()\n");
    inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
    find_original_functions();
    int fd=(*open_ptr)(pathname, flags);
    if (strcmp(pathname, "/proc/uptime")==0)
        opened_fd=fd; // that's our file! record its file descri
ptor
    else
        opened_fd=0;
    return fd;
};

int close(int fd)
{
```

```
    find_original_functions();
    if (fd==opened_fd)
        opened_fd=0; // the file is not opened anymore
    return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
    find_original_functions();
    if (opened_fd!=0 && fd==opened_fd)
    {
        // that's our file!
        return snprintf (buf, count, "%d %d", 0x7fffffff, 0x7fff
ffff)+1;
    };
    // not our file, go to real read() function
    return (*read_ptr)(fd, buf, count);
};
```

源代码

把它编译成动态链接库：

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

运行uptime，并让它在加载其它库之前加载我们的库：

```
LD_PRELOAD=`pwd`/fool_uptime.so uptime
```

可以看到：

```
01:23:02 up 24855 days, 3:14, 3 users, load average: 0.00, 0.01,
 0.05
```

如果LD_PRELOAD环境变量一直指向我们的动态链接库文件名，其它程序在启动的时候也会加载我们的动态链接库。

更多的例子请看：

- Verysimpleinterceptionofthestrcmp()(YongHuang)
- KevinPulo—FunwithLD_PRELOAD.Alotofexamplesandideas.
- File functions interception for compression/decompression files on fly (zlibc).

第六十八章

Windows Nt

68.1 CRT(win32)

程序一开始就从main()函数执行的？事实并非如此。如果我们用IDA或者HIEW打开一个可执行文件，我们可以看到OEP（Original Entry Point）指向了其它代码块。这些代码做了一些维护和准备工作之后再把控制流交给我们的代码。这就是所谓的startup-code或叫CRT code(C RunTime)。

main()函数通过一个数组接收命令行传递过来的参数，环境变量与此类似。通常情况下，传递一个字符串到程序之后，CRT code会用空格来分割它们。CRT code同样也准备了一个envp来存放环境变量。如果是GUI版本的win32程序，入口函数需要使用WinMain()来代替main()函数，它也有自己的参数。

```
int CALLBACK WinMain(
    _In_ HINSTANCE hInstance,
    _In_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine,
    _In_ int nCmdShow
);
```

CRT code同样会准备好它所需要的所有参数。

此外，main()函数的返回值是它的退出码。CRT code将它作为ExitProcess()的参数。

通常，每个编译器都有它自己的CRT code。

下面是MSVC 2008特有的CRT code。

```
___tmainCRTStartup proc near

var_24 = dword ptr -24h
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
ms_exc = CPPEH_RECORD ptr -18h

    push 14h
    push offset stru_4092D0
    call __SEH_prolog4
    mov eax, 5A4Dh
    cmp ds:400000h, ax
    jnz short loc_401096
```

```
    mov eax, ds:40003Ch
    cmp dword ptr [eax+400000h], 4550h
    jnz short loc_401096
    mov ecx, 10Bh
    cmp [eax+400018h], cx
    jnz short loc_401096
    cmp dword ptr [eax+400074h], 0Eh
    jbe short loc_401096
    xor ecx, ecx
    cmp [eax+4000E8h], ecx
    setnz cl
    mov [ebp+var_1C], ecx
    jmp short loc_40109A


loc_401096: ; CODE XREF: ___tmainCRTStartup+18
            ; ___tmainCRTStartup+29 ...
    and [ebp+var_1C], 0

loc_40109A: ; CODE XREF: ___tmainCRTStartup+50
    push 1
    call __heap_init
    pop ecx
    test eax, eax
    jnz short loc_4010AE
    push 1Ch
    call _fast_error_exit
    pop ecx

loc_4010AE: ; CODE XREF: ___tmainCRTStartup+60
    call __mtinit
    test eax, eax
    jnz short loc_4010BF
    push 10h
    call _fast_error_exit
    pop ecx

loc_4010BF: ; CODE XREF: ___tmainCRTStartup+71
    call sub_401F2B
    and [ebp+ms_exc.disabled], 0
    call __ioinit
    test eax, eax
    jge short loc_4010D9
    push 1Bh
    call __amsg_exit
    pop ecx

loc_4010D9: ; CODE XREF: ___tmainCRTStartup+8B
    call ds:GetCommandLineA
    mov dword_40B7F8, eax
    call ___crtGetEnvironmentStringsA
    mov dword_40AC60, eax
    call __setargv
```

```
    test eax, eax
    jge short loc_4010FF
    push 8
    call __amsg_exit
    pop ecx

loc_4010FF: ; CODE XREF: ___tmainCRTStartup+B1
    call __setenvp
    test eax, eax
    jge short loc_401110
    push 9
    call __amsg_exit
    pop ecx

loc_401110: ; CODE XREF: ___tmainCRTStartup+C2
    push 1
    call __cinit
    pop ecx
    test eax, eax
    jz short loc_401123
    push eax
    call __amsg_exit
    pop ecx
loc_401123: ; CODE XREF: ___tmainCRTStartup+D6
    mov eax, envp
    mov dword_40AC80, eax
    push eax ; envp
    push argv ; argv
    push argc ; argc
    call _main
    add esp, 0Ch
    mov [ebp+var_20], eax
    cmp [ebp+var_1C], 0
    jnz short $LN28
    push eax ; uExitCode
    call $LN32

$LN28: ; CODE XREF: ___tmainCRTStartup+105
    call __cexit
    jmp short loc_401186


$LN27: ; DATA XREF: .rdata:stru_4092D0
    mov eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for funct
ion 401044
    mov ecx, [eax]
    mov ecx, [ecx]
    mov [ebp+var_24], ecx
    push eax
    push ecx
    call __XcptFilter
    pop ecx
    pop ecx
```

```
$LN24:
    retn

$LN14: ; DATA XREF: .rdata:stru_4092D0
    mov esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for func
tion 401044
    mov eax, [ebp+var_24]
    mov [ebp+var_20], eax
    cmp [ebp+var_1C], 0
    jnz short $LN29
    push eax ; int
    call __exit

$LN29: ; CODE XREF: ___tmainCRTStartup+135
    call __c_exit

loc_401186: ; CODE XREF: ___tmainCRTStartup+112
    mov [ebp+ms_exc.disabled], 0FFFFFFFEh
    mov eax, [ebp+var_20]
    call __SEH_epilog4
    retn
```

在这里我们看到代码调用了GetCommandLineA()，setargv()和setenvp()去填充argc，argv，envp全局变量。

最后，使用这些参数去调用main()函数。

有些函数调用了与自身类似的函数，如heap_init()，ioinit()。

如果你尝试在CRT code代码中使用malloc()，它将异常退出下面的错误：

```
runtime error R6030
- CRT not initialized
```

在C++中，全局对象的初始化也同样发生在main()函数执行之前的CRT：51.4.1。

main()函数的返回值传给cexit()或$LN32，后者调用doexit()。

能否摆脱CRT？这个当然，如果你知道你在做什么的话。

MSVC的链接器可以通过/ENTRY选项设置入口函数。

```
#include <windows.h>
int main()
{
    MessageBox (NULL, "hello, world", "caption", MB_OK);
};
```

让我们用MSVC 2008来编译它。

```
cl no_crt.c user32.lib /link /entry:main
```

我们可以获得一个大小为2560字节的runnable.exe。它有一个PE头，调用MessageBox的指令，数据段中有两串字符串，而MessageBox函数导入自user32.DLL。

这个程序能够正常运行，但你不能在main()函数里面使用WinMain()的四个参数。准确点来说你能，但是这些参数并没有在执行的时候准备好。

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

它会报一个链接警告：

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image
may not run
```

我们可以获得一个720字节的exe文件。它可以在Windows 7 x86上正常运行，但是没办法在x64上运行（当你运行它的时候会将先是一条错误信息）。更多的优化可能可以提高执行效率，但如你所见，很快就出现了兼容问题。

68.2 Win32 PE

PE是Windows下的可执行文件格式。

.exe，.dll，.sys文件它们之间的区别是，.exe和.sys文件通常没有导出表，只有导入表。

DLL文件和其它PE文件类似，有一个入口点（OEP）（DllMain()函数），但一般情况下很少DLL带有这个函数。

.sys通常是一个设备驱动程序。

作为驱动程序，Windows需要检验它的PE文件并保证它是正确的。

从Windows Vista开始，一个驱动程序文件必须拥有数字签名，否则它会被拒绝加载。

每个PE文件都由一段打印“This program cannot be run in DOS mode.”的DOS程序块开始。如果你的程序运行于DOS或者Windows 3.1（这些OS并不识别PE文件格式），这个DOS程序块将被执行打印。

68.2.1 术语

- Module（模块） - 一个exe/dll文件。
- Process（进程） - 加载到内存中并正在运行的程序，通常由一个exe文件和多个dll文件组成。
- Process memory（进程内存） - 进程所在容所。每个进程都拥有自己的内存。通常是加载的模块，栈内存，堆内存等等。
- VA（虚拟地址） - 可以被程序所使用的地址。
- Base address（基地址） - 模块被加载到进程内存后的地址。
- RVA（相对虚拟地址） - VA地址减去基地址后的地址。PE文件中有许多地址使用RVA地址。
- IAT（导入地址表）- 一个导入符号地址的数组。通常由一个IMAGE_DIRECTORY_ENTRY_IAT数据目录指向IAT。值得注意的是，IDA可会给IAT分配一个名为.idata的pseudo-section，即使IAT是其它section的一部分。
- INT（导入名称表） - 一个导入符号名的数组。

68.2.2 Base address

问题是，模块（DLL）的开发者不可能事先知道哪些地址分配给哪些模块使用的。

这就是为什么两个具有相同基地址的DLL需要一个加载到这个基地址而另外一个加载到进程的其它空闲内存处并调整第二个DLL的虚拟地址。

通常情况下，MSVC链接器生成.exe文件的基地址是0x400000，并把代码段安排在0x401000。这意味着该代码段的RVA地址是0x1000。DLL的基地址通常被MSVC链接器安排在0x10000000。

还有一种情况下加载模块时会导致基地址浮动。

这就是ASLR(Address Space Layout Randomization（地址空间布局随机化）)。

一个shellcode想要执行必须调用到系统的函数。

在老的操作系统当中（如果是WindowsNT，则在Windows Vista之前），系统的DLL（如kernel32.dll，user32.dll）总是加载到已知的地址。如果我们还记得的话，它们的版本是很少有变动的。因为函数的地址是固定的，shellcode可以直接调用它们。

为了避免这种情况，ASLR每次在加载模块的时候都会随机安排它们的基地址。

支持ASLR的程序在PE头中会设置IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE标识表明其支持ASLR。

68.2.3 Subsystem

还有一个subsystem字段, 通常是:

- native (sys驱动程序)
- console (控制台程序)
- GUI (图形程序)

68.2.4 OS version

PE文件还规定了可以加载它的最小Windows版本号。有一个表保存了PE的版本号和相应的Windows开发代号。

举个例子，MSVC 2005编译的.exe文件运行在Windows NT4（version 4.00）。但MSVC 2008不是（生成文件的版本是5.00，至少运行于Windows 2000）。

MSVC 2012生成的.exe文件默认是6.00版本，最低平台要求至少是Windows Vista。但可以通过更改编译选项，强制编译器支持Windows XP。

68.2.5 Sections

一部分section似乎存在于所有可执行文件格式里面。

下面的标志位用于区分代码和常量数据：

- 当IMAGE_SCN_CNT_CODE或IMAGE_SCN_MEM_EXECUTE被置位，表示该section是一个可执行代码。
- 在数据section中，IMAGE_SCN_CNT_INITIALIZED_DATA，IMAGE_SCN_MEM_READ和IMAGE_SCN_MEM_WRITE被置位。
- 在未初始化section和空section中，IMAGE_SCN_CNT_UNINITIALIZED_DATA， IMAGE_SCN_MEM_READ和IMAGE_SCN_MEM_WRITE被置位。
- 在常量数据section（写保护）中，IMAGE_SCN_CNT_INITIALIZED_DATA和IMAGE_SCN_MEM_READ被置位，但不可以置位IMAGE_SCN_MEM_WRITE。当一个进程尝试在这个section写数据时，进程会崩溃掉。

每个section在PE文件可能有一个名字，但是它并不是很重要。通常（但不总是）代码section的名字是.text，数据section是.data，常量数据section是.rdata(readable data)。其它流行的名字还有：

- .idata—imports section（导入section）。IDA可能会创建一个类似(68.2.1)的pseudo-section。
- .edata—exports section（导出section）。
- .pdata—在Windows NT（MIPS，IA64，x64）包含了所有异常信息。
- .reloc—relocs section（重定位section）
- .bss—uninitialized data（未初始化数据（BSS））
- .tls—thread local storage（线程局部存储（TLS））
- .rsrc—resources（资源）
- .CRT—可能存在古老的MSVC版本编译出来的二进制文件里面。

PE文件的打包器/加密器经常打乱section名字或者把名字替换为自己的。

MSVC允许你任意命名section。

一些编译器和链接器可以添加一个用于调试符号和其他调试信息的section(例如MinGW)。但不包括MSVC现在的版本(提供单独的PDB文件用于这个目的)。

这是PE文件的section结构体定义：

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

一些相关的字段的解释：PointerToRawData是在磁盘文件中的偏移，VirtualAddress在Hiew中是装载到内存中的RVA。

68.2.6 Relocations (relocs)

也称为FIXUP-s（在Hiew）。

他们也存在于几乎所有的可执行文件格式。

显然，模块可以被加载到各种基地地址，但如何处理全局变量？一个解决方案是使用位置无关代码（67.1章），但它并不是总是有用的。

这就是重定位表存在的理由：当模块加载到不同的基地址的时候，它们的入口地址都需要修正。

举个例子，有一个全局变量的地址是0x410000，它是这样访问的：

```
A1 00 00 41 00    mov eax, [000410000]
```

模块的基地址是0x400000，全局变量的RVA地址是0x10000。

如果模块加载到0x500000这个基地址，那么全局变量实际的地址必须是0x510000。

我们可以看到，在0xA1字节之后，变量的地址编码到MOV指令中的。

这就是为什么0xA1字节之后的4个字节地址写在了重定位表。

如果模块加载到不同的基地址，操作系统加载器枚举重定位表中所有地址，查找每个32位的地址，减去原来的基地址(我们这里得到了RVA)，并添加新的基地址。

如果模块加载到原来的基地址，那么不做任何事情。

所有的全局变量都可以这样处理。

重定位表可能有各种类型，但是在x86处理器的Windows中，通常是IMAGE_REL_BASED_HIGHLOW。

顺便说一下，重定位表在Hiew是隐藏的。相关例子请查看（Figure 7.12）。

OllyDbg会用下划线标识哪些使用了重定位表。相关例子请查看（Figure 13.11）。

68.2.7 Exports and imports

众所周知，任何可执行文件都必须使用操作系统提供的服务和其它一些动态链接库。

可以说，一个模块（通常是DLL）的函数通常都是导出提供给其它模块使用（.exe文件或其它DLL）。

这种情况下，每个DLL都有一个导出（exports）表，由模块的函数加它们的地址组成。

每个exe或dll文件也有一个导入（imports）表，里面包含了程序执行所需函数对应的DLL文件名。

在加载main.exe文件之后，操作系统加载器开始处理导入表：它加载所需的DLL文件，接着在DLL的导入表查找对应函数名字的地址，然后把它们的地址写到main.exe模块的IAT（（Import Address Table）导入表）。

我们可以看到，加载器必须大量比较函数名，但字符串比较效率并不是很高。所以有一个支持“ordinals”或“hints”的东西，表示函数存储在表中的序号，用于代替它们的函数名。

这使得它们可以更快地加载DLL。Ordinals在导出表中永远都存在。

举个例子：一个使用MFC库的程序都是通过ordinals加载mfc*.dll，在这种程序中，INT（Import Name Table）是不存在MFC函数名字的。

使用IDA加载这类程序的时候，如果告诉它mfc*.dll文件路径，则可以看到函数名。如果不告诉IDA这些DLL路径，它会显示诸如mfc80_123而不是函数名。

Imports section

编译器通常会给导入表及其相关内容分配一个单独的section（名字类似.idata），但这不是一个强制规定。

因为术语混乱，导入表是一个比较令人困惑的地方。让我们尝试一下整理这些信息。

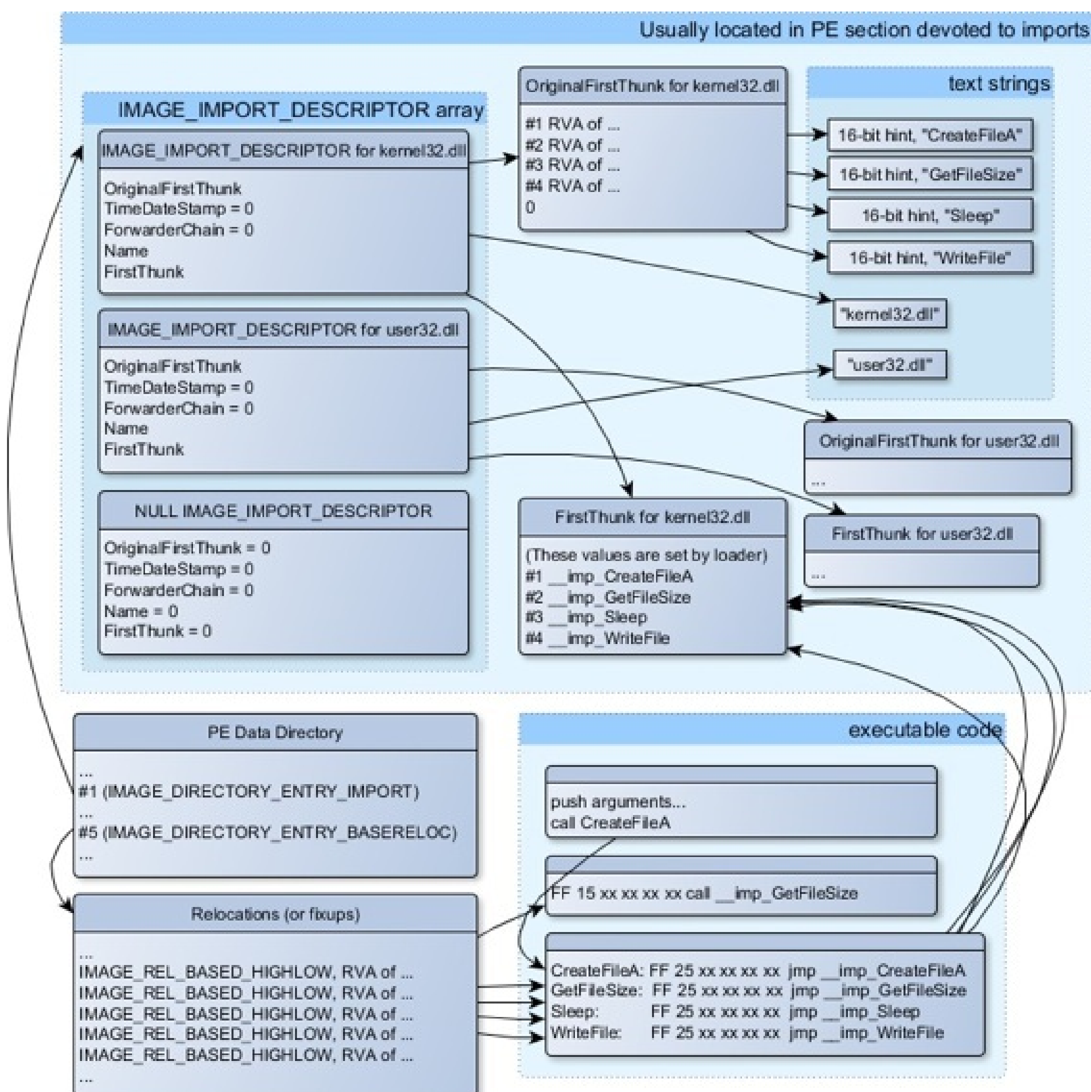


Figure 68.1: A scheme that unites all PE-file structures related to imports

里面主要的结构是IMAGE_IMPORT_DESCRIPTOR数组。每个被加载进来的DLL占用一个元素。

每个元素包含一个文本字符串（DLL名字）的RVA地址。

OriginalFirstThink是INT表的RVA地址。这是一个RVA地址的数组，里面每个成员都指向一个函数名的文本字符串。每个函数名的字符串之前是一个16位的("hint")-"ordinal"整数。

加载的时候，如果可以通过ordinal找到函数，那么就不需要使用字符串比较来查找函数。数组的最后一个元素是0。还有一个FirstThunk字段指向IAT表，这个地方是加载器重写需要重新解析函数的地址的RVA地址。

需要加载器重写地址的函数在IDA中加了诸如这种标记：__imp_CreateFileA。

加载器至少有两种方法重写地址：

- 代码会有诸如调用__imp_CreateFileA的指令，因为导入函数的地址在某种意义上是一个全局变量，当模块加载到不同的基地址时，call指令的地址被添加到重定位表中。 但是，显然这种方法可能会扩大重定位表。因为有可能从这个

模块大量调用导入的函数。而且，重定位表太大的话会减慢模块的加载速度。

- 每个导入函数给它分配一条jmp指令，使用jmp指令加上重定位表的地址跳转到导入函数。这些入口点被称之为“thunks”，所有调用导入函仅需要调用相对应的“thunk”，这种情况下不需要额外的重定位操作，因为这些CALL都使用相对地址，不需要额外的调整操作。

这两种方法可以组合使用。可能的话，链接器给那些被调用太多次的函数创建一个“thunk”，然而默认情况下不是这样。

顺便说一下，FirstThunk指向的函数地址数组不必要位于IAT section。举个例子，我曾经写的PE_add_import工具可以给.exe文件添加一个导入函数。在早些时候，这个工具可以让你的函数调用其它DLL文件的函数。我的工具添加了类似下面的代码：

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

FirstThunk指向第一条指令，换句话说，当加载yourdll.dll的时候，加载器在代码中写入function函数的正确地址。

还值得注意的是代码段通常是写保护的，因此我的工具在code section添加了一个IMAGE_SCN_MEM_WRITE标志位。否则，程序在加载的时候会爆出错误码为5（访问失败）的异常错误。

有人可能会问：如果我提供一个程序与一组不变的DLL文件，是有可能加快加载过程？

是的，它可以提前把函数的地址写入到导入表的FirstThunk数组。IMAGE_IMPORT_DESCRIPTOR结构有一个Timestamp字段。如果这个变量存在，则加载器会比较这个变量和DLL文件日期时间。如果它们相等，那么加载器不做任何事情，所以加载过程可以很快完成。这就是所谓的“old-style binding”。为了加快程序的加载，Matt Pietrek. “An In-Depth Look into the Win32 Portable Executable File Format”，建议你的程序安装在最终用户的计算机后不久做捆绑。

PE文件的打包器/加密器也可以压缩/加密导入表。在这种情况下，Windows的加载器当然不会加载所有需要的DLL。因此打包器/加密器只能通过LoadLibrary()和GetProcAddress()来获取所需函数。

安装在Windows系统中的标准DLL文件，IAT往往是位于PE文件的开头。据说，这是一种优化。加载时.exe文件不是全部加载到内存，它是“映射”和加载部分需要被访问到的内存。可能微软的开发者认为这样加载比较快。

68.2.8 Resources

资源在PE文件只是一组图标，图片，文本字符串，对话框描述。因为把它们从主代码分离了出来，所以多国语言程序很容易实现，只需要根据操作系统设置的语言去选择文本或图片的语言。

作为一个副作用，通过使用诸如ResHack的编辑器，即使在没有专业知识的情况下，也可以轻松地编辑和保存可执行文件的资源。

68.2.9 .NET

.NET的程序并不编译成机器码，而是编译成字节码。严格地说，是在.exe文件里面使用字节码代替x86机器。然而，进入入口点（OEP）还是需要一小段x86机器码:

```
jmp mscoree.dll!_CorExeMain
```

.NET的加载器位于mscoree.dll，由它来处理PE文件。它存在于之前的所有Windows XP操作系统。从XP启动的时候，OS的加载器能够探测.NET文件并通过JMP指令执行。

68.2.10 TLS

这个section包含了初始化TLS的数据（65章）（如果需要的话）。当一个新线程启动的时候，它的TLS数据使用这个section的数据进行初始化。

除此之外，PE文件规范还提供了TLS的初始化！当section，TLS callbacks存在，它们会在传递控制权到主入口点（OEP）之前被调用。这个功能广泛用于PE文件的打包和加密。

68.2.11 工具

- objdump - cygwin版本可以反汇编PE文件
- Hiew - (参考73章)
- pefile - 一个处理PE文件的Python库
- ResHack AKA Resource Hacker — 资源编辑器
- PE_add_import — 添加符号到导入表的简易工具
- PE_patcher — 修补PE文件的简易工具
- PE_search_str_refs — 查找函数在PE文件里对应的字符串的简易工具

68.2.12 扩展阅读

Daniel Pistelli — The .NET File Format

68.3 Windows SEH

68.3.1 让我们先忘了MSVC

在Windows，SEH（Structured Exception Handling（结构化异常处理））是异常处理的一种机制。然而，它是语言无关的，不管是 `C++` 或者其它OOP语言。我们可以看到SEH（从 `C++` 和MSVC扩展）是独立实现的。

每个运行的进程都有一个SEH处理链，TIB有它最后的处理程序的地址。当异常发生时(除零，错误的地址访问，用户通过调用RaiseException()函数引发异常)，操作系统在TIB找到最后的处理程序并调用它，获取异常时CPU的状态信息（如寄存器的值等等）。处理程序当前的异常能否修复，如果能，则修复该异常。如果不能，它通知操作系统无法处理它并由操作系统调用异常处理链中的下一个处理程序,直到处理程序能够处理的异常被发现。

在异常处理链的结尾处有一个标准的处理程序，它显示一个对话框用于通知用户进程崩溃，然后把一些崩溃时CPU的状态信息，收集起来并将其发送给微软开发商。



Figure 68.2: Windows XP

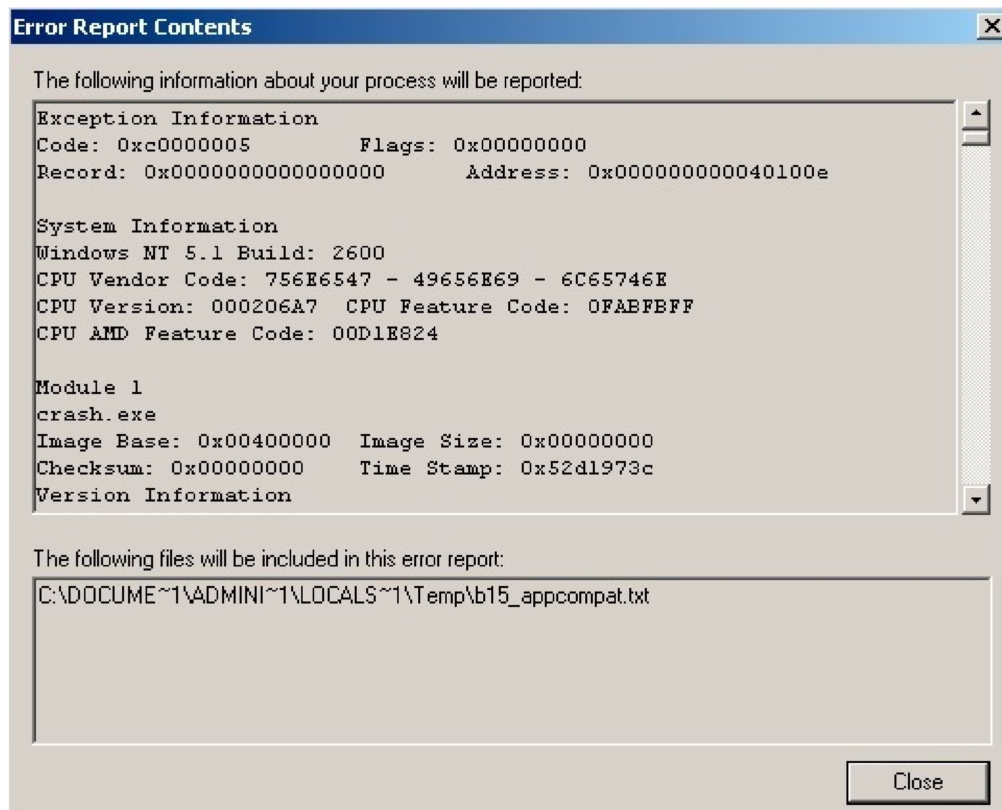


Figure 68.3: Windows XP

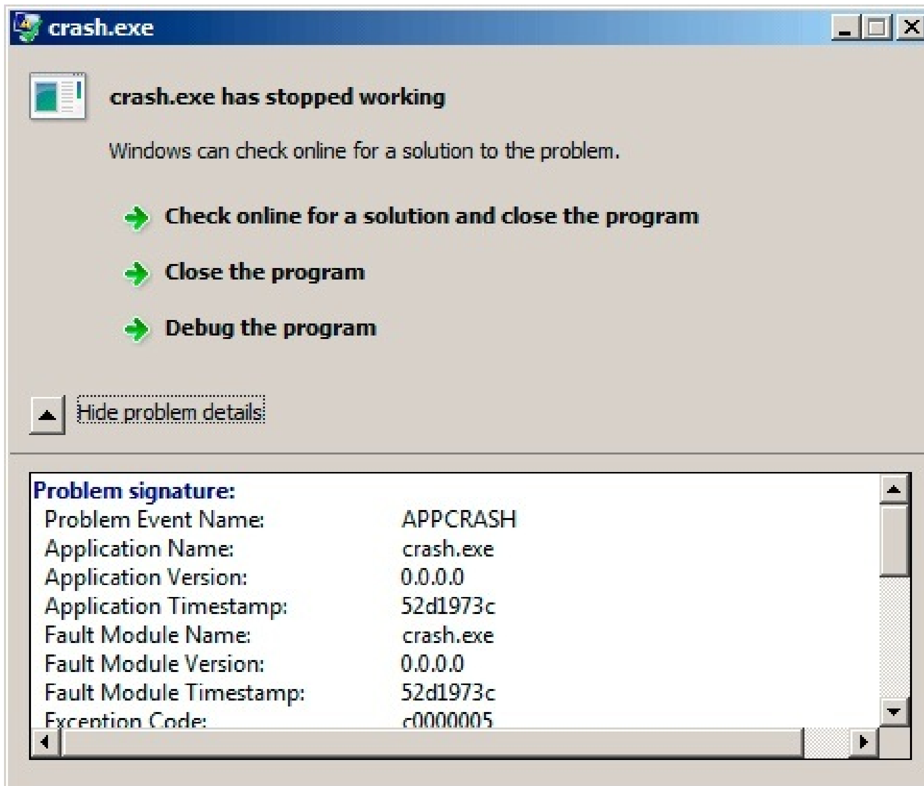


Figure 68.4: Windows 7

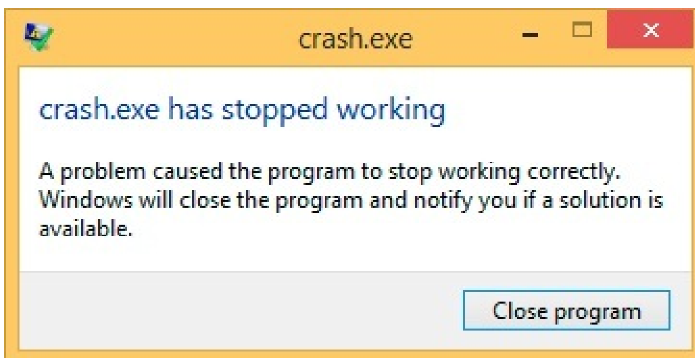


Figure 68.5: Windows 8.1

早些时候，这个处理程序被称为Dr.Watson。

顺便说一句，有些开发人员会在自己的处理程序发送程序崩溃的信息。通过SetUnhandledExceptionFilter()函数注册异常处理程序，如果操作系统没有任何其它方式处理异常，则调用它。一个例子是Oracle RDBMS，它保存了CPU所有可能有用的信息和内存状态的巨大转储文件。

让我们写一个自己的primitive exception handler：

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
```

```
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
    unsigned i;

    printf ("%s\n", __FUNCTION__);
    printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRe
cord->ExceptionCode);
    printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionR
ecord->ExceptionFlags);
    printf ("ExceptionRecord->ExceptionAddress=0x%p\n", Exceptio
nRecord->ExceptionAddress);
    if (ExceptionRecord->ExceptionCode==0xE1223344)
    {
        printf ("That's for us\n");
        // yes, we "handled" the exception
        return ExceptionContinueExecution;
    }
    else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VI
OLATION)
    {
        printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Ea
x);
        // will it be possible to 'fix' it?
        printf ("Trying to fix wrong pointer address\n");
        ContextRecord->Eax=(DWORD)&new_value;
        // yes, we "handled" the exception
        return ExceptionContinueExecution;
    }
    else
    {
        printf ("We do not handle this\n");
        // someone else's problem
        return ExceptionContinueSearch;
    };
}
int main()
{
    DWORD handler = (DWORD)except_handler; // take a pointer to
our handler
    // install exception handler
    __asm
    { // make EXCEPTION_REGISTRATION record:
        push handler // address of handler function
        push FS:[0] // address of previous handler
        mov FS:[0],ESP // add new EXECEPTION_REGISTRATION
    }
    RaiseException (0xE1223344, 0, 0, NULL);
    // now do something very bad
    int* ptr=NULL;
    int val=0;
    val=*ptr;
```

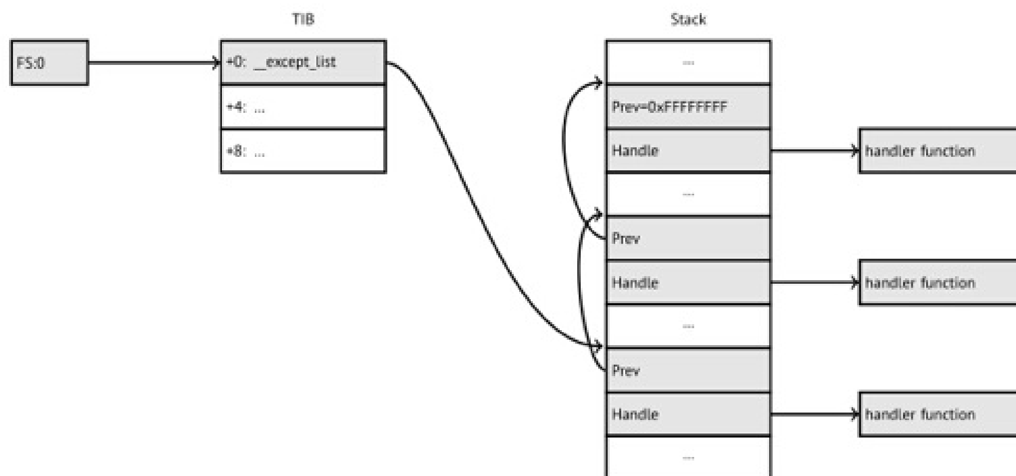
```
    printf ("val=%d\n", val);
    // deinstall exception handler
    __asm
    { // remove our EXECEPTION_REGISTRATION record
        mov eax,[ESP] // get pointer to previous record
        mov FS:[0], EAX // install previous record
        add esp, 8 // clean our EXECEPTION_REGISTRATION off stac
k
    }
    return 0;
}
```

FS段寄存器：在Win32指向TIB。在TIB的第一个元素是指向异常处理指针链中的最后一个处理程序，我们将自己的异常处理程序的地址保存在这里。异常处理链的结点结构体名字是_EXCEPTION_REGISTRATION，这是一个单链表实现的栈容器。

Listing 68.1: MSVC/VC/crt/src/exsup.inc

```
\_EXCEPTION\_REGISTRATION struc
    prev dd ?
    handler dd ?
\_EXCEPTION\_REGISTRATION ends
```

每个结点的handler字段指向一个异常处理程序，每个结点的prev字段指向在栈中的上一个结点。最后一个结点的prev指向0xFFFFFFFF(-1)。



我们的处理程序安装后，我们调用RaiseException()。这是一个用户异常。处理程序检查异常代码，如果异常代码是0xE1223344，它返回ExceptionContinueExecution。这意味着处理程序修复了CPU的状态（通常是EIP/ESP寄存器），操作系统可以恢复运行。如果你稍微修改一下代码，处理程序返回ExceptionContinueSearch，那么操作系统将调用下一个处理程序，如果没有找到处理程序（因为没人捕获该异常），你会看到标准的Windows进程崩溃对话框。

系统异常和用户异常之间的区别是什么？这里有系统的：

as defined in WinBase.h	as defined in ntstatus
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLAT
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISA
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORM
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFL
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERF
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVID
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVER
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INS
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERRO
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRU
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUAB
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFL
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOS
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDL
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEAD
CONTROL_C_EXIT	STATUS_CONTROL_C_EX

这些异常码的定义规则是：

31	29	28	27 ~ 16	15 ~ 0
S	U	0	Facility code	Error code

S是一个基本代码: 11—error; 10—warning; 01—informational; 00—success；U表示是否是用户代码。

这就是为什么我选择了0xE1223344，0xE(1110b)意味着1）user exception（用户异常）;2）error（错误）。

当我们尝试读取地址为0的内存时。因为这个地址在win32中并不被使用，所以会引发一个异常。通过检查异常码是否等于EXCEPTION_ACCESS_VIOLATION常量。

读0地址内存的代码看起来像这样：

Listing 68.2: MSVC 2010

```
...
    xor eax, eax
    mov eax, DWORD PTR [eax] ; exception will occur here
    push eax
    push OFFSET msg
    call _printf
    add esp, 8
...
```

能否修复“on the fly”这个错误然后继续执行程序？当然，我们的异常处理程序可以修复EAX值然后让操作系统继续执行下去。这是我们该做的。printf()将打印1234，因为我们的处理程序执行后EAX不是0，而是全局变量new_value的地址。

若内存管理器有一个关于CPU的错误信号，CPU会暂停线程，在Windows内核查找异常处理程序，然后一个一个调用SEH链的handler。

我在这里使用MSVC 2010，当然，没有任何保证EAX将用于这个指针。

这个地址替换的技巧非常的漂亮，我经常使用它插入到SEH内部中。不过，我忘记了在哪里用它修复“on the fly”错误。

为什么SHE相关的记录存储在栈上而不是其它地方？据说这是因为操作系统不需要在函数执行完成之后关心这些信息。但我不能100%肯定。这有点类似alloca()。

68.3.2 现在让我们回到MSVC

据说，微软的程序员需要在C语言而不是 `C++` 上使用异常，所以它们在MSVC上添加了一个非标准的C扩展。它与 `C++` 的异常没有任何关联。

```
__try
{
    ...
}
__except(filter code)
{
    handler code
}
```

“Finally”块也许能代替handler code：

```
__try
{
    ...
}
__finally
{
    ...
}
```

filte code是一个表达式，告诉handler code是否对应引发的异常。如果你的filte code太大而无法使用一个表达式，可以定义一个单独的filte函数。

在Windows内核有很多这样的结构，下面是几个例子（WRK（Windows Research Kernel））：

Listing 68.3: WRK-v1.2/base/ntos/ob/obwait.c

```
try {
    KeReleaseMutant( (PKMUTANT)SignalObject,
                     MUTANT_INCREMENT,
                     FALSE,
                     TRUE );
} except((GetExceptionCode () == STATUS_ABANDONED ||
          GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
          EXCEPTION_EXECUTE_HANDLER :
          EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();
    goto WaitExit;
}
```

Listing 68.4: WRK-v1.2/base/ntos/cache/cachesub.c

```
try {
    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                  UserBuffer,
                  MorePages ?
                 (PAGE_SIZE - PageOffset) :
                 (ReceivedLength - PageOffset) );
} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
Status ) ) {
```

这里是一个filter code的例子：

Listing 68.5: WRK-v1.2/base/ntos/cache/copysup.c

```
LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
)

/*++

Routine Description:
    This routine serves as a exception filter and has the specia
l job of
    extracting the "real" I/O error when Mm raises STATUS_IN_PAG
E_ERROR
    beneath us.
Arguments:
    ExceptionPointer - A pointer to the exception record that co
ntains
    the real Io Status.
    ExceptionCode - A pointer to an NTSTATUS that is to receive
the real
    status.
Return Value:
    EXCEPTION_EXECUTE_HANDLER

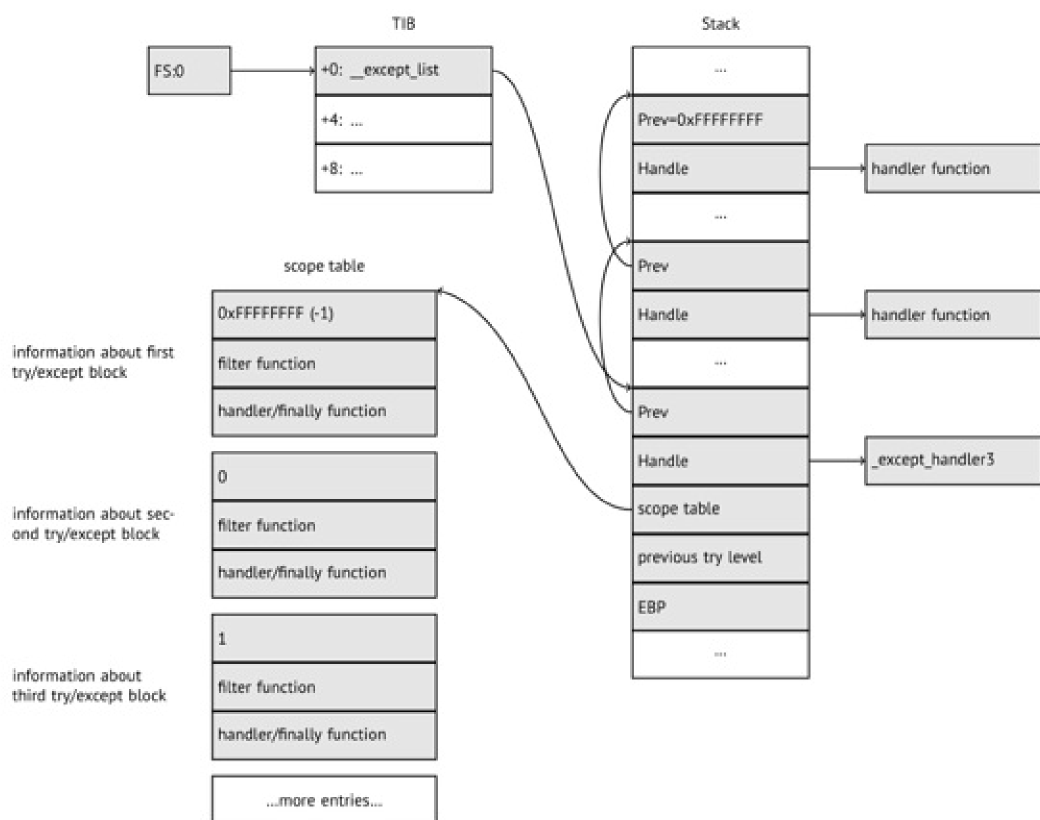
--*/
{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->Exceptio
nCode;
    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
         (ExceptionPointer->ExceptionRecord->NumberParameters >=
 3) ) {
        *ExceptionCode = (NTSTATUS) ExceptionPointer->ExceptionR
ecord->ExceptionInformation[2];
    }
    ASSERT( !NT_SUCCESS(*ExceptionCode) );
    return EXCEPTION_EXECUTE_HANDLER;
}
```

在内部，SEH是操作系统支持的异常扩展。但是处理函数是_except_handler3（对于SEH3）或_except_handler4（对于SEH4）。这个处理函数的代码是与MSVC相关的，它位于它的库或在msvcr*.dll文件。其他的Win32编译器可以提供与之完全不同的机制。

SEH3

SEH3有一个_except_handler3处理函数，而且扩展了_EXCEPTION_REGISTRATION表，并添加了一个指向scope table和previous try level变量。SEH4扩展了scope table缓冲溢出保护。

scope table是一个表，包含了指向filter和handler code的块和每个try/except嵌套。



再者，操作系统只关心prev/handle字段。_except_handler3函数的工作是读取其他字段和scope table，并决定由哪些处理程序来执行。

_except_handler3函数的源代码是闭源的。然而，Sanos操作系统的win32兼容性层重新实现相同的功能。其它类似的实现有Wine和ReactOS。

如果filter指针为NULL，handler指针则指向finally代码块。

执行期间，栈中的previous try level变量发生变化，所以_except_handler3可以获取当前嵌套级的信息，才知道要使用scope table哪一表项。

SEH3: 一个try/except块例子

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>
int main()
{
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
        *p = 13; // causes an access violation exception;
        printf("hello #2!\n");
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEAR
CH)
    {
        printf("access violation, can't recover\n");
    }
}
```

Listing 68.6: MSVC 2003

```
$SG74605 DB 'hello #1!', 0aH, 00H
$SG74606 DB 'hello #2!', 0aH, 00H
$SG74608 DB 'access violation, can''t recover', 0aH, 00H
_DATA ENDS

; scope table

CONST SEGMENT
$T74622 DD 0ffffffffH   ; previous try level
    DD FLAT:$L74617     ; filter
    DD FLAT:$L74618     ; handler
CONST ENDS

_TEXT SEGMENT
$T74621 = -32     ; size = 4
_p$ = -28         ; size = 4
__$SEHRec$ = -24  ; size = 24
_main PROC NEAR
    push ebp
    mov ebp, esp
    push -1                            ; previous try level
    push OFFSET FLAT:$T74622           ; scope table
    push OFFSET FLAT:__except_handler3 ; handler
    mov eax, DWORD PTR fs:__except_list
    push eax                           ; prev
    mov DWORD PTR fs:__except_list, esp
    add esp, -16
    push ebx                           ; saved 3 registers
    push esi                           ; saved 3 registers
```

```
    push edi                             ; saved 3 registers
    mov DWORD PTR __$SEHRec$[ebp], esp
    mov DWORD PTR _p$[ebp], 0
    mov DWORD PTR __$SEHRec$[ebp+20], 0   ; previous try level
    push OFFSET FLAT:$SG74605             ; 'hello #1!'
    call _printf
    add esp, 4
    mov eax, DWORD PTR _p$[ebp]
    mov DWORD PTR [eax], 13
    push OFFSET FLAT:$SG74606             ; 'hello #2!'
    call _printf
    add esp, 4
    mov DWORD PTR __$SEHRec$[ebp+20], -1  ; previous try level
    jmp SHORT $L74616
    ; filter code
$L74617:
$L74627:
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov edx, DWORD PTR [ecx]
    mov eax, DWORD PTR [edx]
    mov DWORD PTR $T74621[ebp], eax
    mov eax, DWORD PTR $T74621[ebp]
    sub eax, -1073741819; c0000005H
    neg eax
    sbb eax, eax
    inc eax
$L74619:
$L74626:
    ret 0
    ; handler code
$L74618:
    mov esp, DWORD PTR __$SEHRec$[ebp]
    push OFFSET FLAT:$SG74608             ; 'access violation, c
an''t recover'
    call _printf
    add esp, 4
    mov DWORD PTR __$SEHRec$[ebp+20], -1  ; setting previous try
 level back to -1
$L74616:
    xor eax, eax
    mov ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov DWORD PTR fs:__except_list, ecx
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
_TEXT ENDS
END
```

在这里我们可以看到SEH帧是如果在栈中构建出来的，scope table位于CONST segment-事实上，这些字段是不被改变的。一件有趣的事情是如何改变previous try level变量。它的初始化值是0xFFFFFFFF(-1)。当进入try语句块的时候，变量赋值为0。当try语句块结束的时候，写回-1。我们还能看到filter和handler code的地址。因此，我们可以很容易在函数里看到try/except是如何构造的。

由于函数序言的SEH安装代码被多个函数共享，有时候编译器会在函数序言插入调用SEH_prolog()函数，这就完成了这个任务。该SEH回收代码是SEH_epilog()函数。

让我们尝试用tracer运行这个例子：

```
tracer.exe -l:2.exe --dump-seh
```

Listing 68.7: tracer.exe output

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) Excepti
onInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!_e
xcept_handler
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.e
xe!main+0x60) handler=0x401088 (2.exe!main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!_e
xcept_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.e
xe!mainCRTStartup+0x18d) handler=0x401545 (2.exe!mainCRTStartup+
0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.
dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header: GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (n
tdll.dll!___safe_se_handler_table+0x20) handler=0x771f90eb (ntdl
l.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdl
l.dll!_FinalExceptionHandler@16)
```

我们看到，SEH链包含4个handler。

前面两个是我们的例子。两个？但是我们只有一个？是的，一个是CRT的_mainCRTStartup()函数设置的。并至少作为FPU异常的处理。它的源码可以在MSVC的安装目录找到：crt/src/winxfltr.c。

第三个是ntdll.dll的SEH4，第四个handler也位于ntdll.dll，跟MSVC没什么关系，它有一个自描述函数名。

正如你所见，在一个链中有三种类型的处理函数：一个跟MSVC(最后一个)没什么关系和两个与MSVC关联的：SEH3和SEH4。

SEH3: 两个try/except块例子

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>
int filter_user_exceptions (unsigned int code, struct _EXCEPTION
_POINTERS *ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}
int main()
{
    int* p = NULL;
    __try
    {
        __try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
            printf ("0x112233 raised. now let's crash\n");
            *p = 13; // causes an access violation exception;
        }
        __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION
?
                 EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_
SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(), GetExcep
tionInformation()))
    {
        // the filter_user_exceptions() function answering to th
e question
        // "is this exception belongs to this block?"
        // if yes, do the follow:
        printf("user exception caught\n");
    }
}
```

现在有两个try块，所以scope table现在有两个元素，每个块占用一个。Previous try level随着try块的进入或退出而改变。

Listing 68.8: MSVC 2003

```
$SG74606 DB 'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB 'yes, that is our exception', 0aH, 00H
$SG74610 DB 'not our exception', 0aH, 00H
$SG74617 DB 'hello!', 0aH, 00H
$SG74619 DB '0x112233 raised. now let''s crash', 0aH, 00H
$SG74621 DB 'access violation, can''t recover', 0aH, 00H
$SG74623 DB 'user exception caught', 0aH, 00H
_code$ = 8 ; size = 4
_ep$ = 12 ; size = 4
_filter_user_exceptions PROC NEAR
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _code$[ebp]
    push eax
    push OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call _printf
    add esp, 8
    cmp DWORD PTR _code$[ebp], 1122867; 00112233H
    jne SHORT $L74607
    push OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call _printf
    add esp, 4
    mov eax, 1
    jmp SHORT $L74605
$L74607:
    push OFFSET FLAT:$SG74610 ; 'not our exception'
    call _printf
    add esp, 4
    xor eax, eax
$L74605:
    pop ebp
    ret 0
_filter_user_exceptions ENDP

    ; scope table

CONST SEGMENT
$T74644 DD 0ffffffffH ; previous try level for outer block
    DD FLAT:$L74634 ; outer block filter
    DD FLAT:$L74635 ; outer block handler
    DD 00H ; previous try level for inner block
    DD FLAT:$L74638 ; inner block filter
    DD FLAT:$L74639 ; inner block handler
CONST ENDS

$T74643 = -36 ; size = 4
$T74642 = -32 ; size = 4
_p$ = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24
_main PROC NEAR
```

```
    push ebp
    mov ebp, esp
    push -1 ; previous try level
    push OFFSET FLAT:$T74644
    push OFFSET FLAT:__except_handler3
    mov eax, DWORD PTR fs:__except_list
    push eax
    mov DWORD PTR fs:__except_list, esp
    add esp, -20
    push ebx
    push esi
    push edi
    mov DWORD PTR __$SEHRec$[ebp], esp
    mov DWORD PTR _p$[ebp], 0
    mov DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block entere
d. set previous try level to 0
    mov DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block entere
d. set previous try level to 1
    push OFFSET FLAT:$SG74617 ; 'hello!'
    call _printf
    add esp, 4
    push 0
    push 0
    push 0
    push 1122867 ; 00112233H
    call DWORD PTR __imp__RaiseException@16
    push OFFSET FLAT:$SG74619 ; '0x112233 raised. now let''s cra
sh'
    call _printf
    add esp, 4
    mov eax, DWORD PTR _p$[ebp]
    mov DWORD PTR [eax], 13
    mov DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited
. set previous try level back to 0
    jmp SHORT $L74615
    ; inner block filter
$L74638:
$L74650:
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov edx, DWORD PTR [ecx]
    mov eax, DWORD PTR [edx]
    mov DWORD PTR $T74643[ebp], eax
    mov eax, DWORD PTR $T74643[ebp]
    sub eax, -1073741819; c0000005H
    neg eax
    sbb eax, eax
    inc eax
$L74640:
$L74648:
    ret 0
    ; inner block handler
$L74639:
    mov esp, DWORD PTR __$SEHRec$[ebp]
```

```
    push OFFSET FLAT:$SG74621 ; 'access violation, can''t recove
r'
    call _printf
    add esp, 4
    mov DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited
. set previous try level back to 0
$L74615:
    mov DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block exite
d, set previous try level back to -1
    jmp SHORT $L74633
    ; outer block filter
$L74634:
$L74651:
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov edx, DWORD PTR [ecx]
    mov eax, DWORD PTR [edx]
    mov DWORD PTR $T74642[ebp], eax
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
    push ecx
    mov edx, DWORD PTR $T74642[ebp]
    push edx
    call _filter_user_exceptions
    add esp, 8
$L74636:
$L74649:
    ret 0
    ; outer block handler
$L74635:
    mov esp, DWORD PTR __$SEHRec$[ebp]
    push OFFSET FLAT:$SG74623 ; 'user exception caught'
    call _printf
    add esp, 4
    mov DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks exite
d. set previous try level back to -1
$L74633:
    xor eax, eax
    mov ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov DWORD PTR fs:__except_list, ecx
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
```

如果我们在handler中调用的printf()函数设置一个断点，可以看到另一个SEH handler如何被添加。同样，我们还可以看到scope table包含两个元素。

```
tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh
```

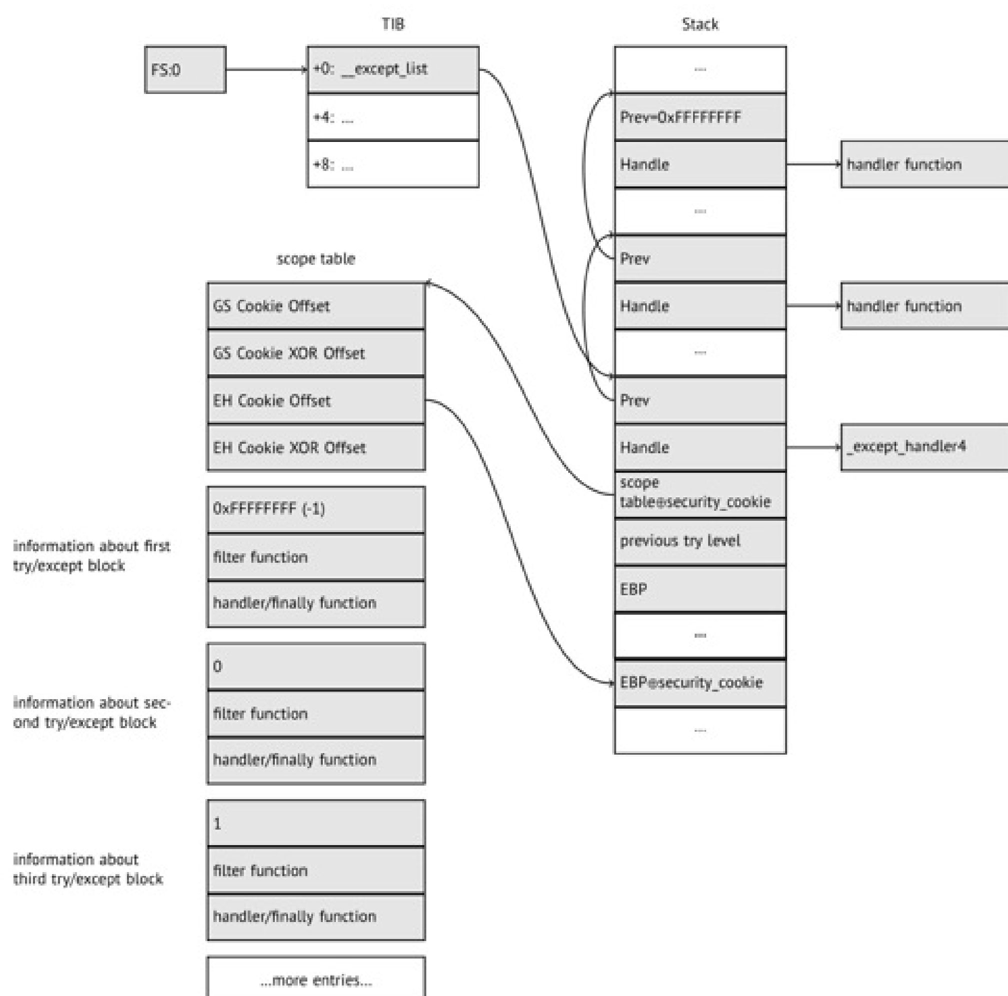
Listing 68.9: tracer.exe output

```
(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.
dll!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!_e
xcept_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.e
xe!main+0xb0) handler=0x40113b (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.ex
e!main+0x78) handler=0x401100 (3.exe!main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!_e
xcept_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.e
xe!mainCRTStartup+0x18d) handler=0x401621 (3.exe!mainCRTStartup+
0x1a1
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.
dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header: GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (n
tdll.dll!___safe_se_handler_table+0x20) handler=0x771f90eb (ntdl
l.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdl
l.dll!_FinalExceptionHandler@16)
```

SEH4

在缓冲区攻击期间（18.2章），scope table的地址可以被重写。所以从MSVC 2005开始，SEH3升级到SEH4后有了缓冲区溢出保护。现在scope table指针与一个security cookie（一个随机值）做异或运算。scope table扩展了包含两个指向security cookie指针的头部。每个元素都有另一个栈内偏移值：栈帧的地址（EBP）与security_cookie异或。该值将在异常处理过程中读取并检查其正确性。栈中的security cookie每次都是随机的，所以远程攻击者无法预测到它。

SEH4的previous try level初始化值是-2而不是-1。



这里有两个使用MSVC编译的SEH4例子：

Listing 68.10: MSVC 2012: one try block example

```
$SG85485 DB    'hello #1!', 0aH, 00H
$SG85486 DB    'hello #2!', 0aH, 00H
$SG85488 DB    'access violation, can''t recover', 0aH, 00H

; scope table:
xdata$x          SEGMENT
__sehtable$_main DD 0fffffffeH   ; GS Cookie Offset
    DD           00H             ; GS Cookie XOR Offset
    DD           0ffffffccH      ; EH Cookie Offset
    DD           00H             ; EH Cookie XOR Offset
    DD           0fffffffeH      ; previous try level
    DD           FLAT:$LN12@main ; filter
    DD           FLAT:$LN8@main  ; handler
xdata$x          ENDS

$T2 = -36        ; size = 4
_p$ = -32        ; size = 4
tv68 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
```

```
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to
scope table
    xor    eax, ebp
    push   eax                              ; ebp ^ security_coo
kie
    lea    eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to VC_EXCE
PTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push   OFFSET $SG85485 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET $SG85486 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp    SHORT $LN6@main

; filter:
$LN7@main:
$LN12@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    cmp    DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN4@main
    mov    DWORD PTR tv68[ebp], 1
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR tv68[ebp], 0
$LN5@main:
    mov    eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
    ret    0
```

```
; handler:
$LN8@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85488 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Listing 68.11: MSVC 2012: two try blocks example

```
$SG85486 DB    'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB    'yes, that is our exception', 0aH, 00H
$SG85490 DB    'not our exception', 0aH, 00H
$SG85497 DB    'hello!', 0aH, 00H
$SG85499 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG85501 DB    'access violation, can''t recover', 0aH, 00H
$SG85503 DB    'user exception caught', 0aH, 00H

xdata$x    SEGMENT
__sehtable$_main DD 0fffffffeH         ; GS Cookie Offset
                 DD    00H             ; GS Cookie XOR Offset
                 DD    0ffffffc8H      ; EH Cookie Offset
                 DD    00H             ; EH Cookie Offset
                 DD    0fffffffeH      ; previous try level for
outer block
                 DD    FLAT:$LN19@main ; outer block filter
                 DD    FLAT:$LN9@main  ; outer block handler
                 DD    00H             ; previous try level for
inner block
                 DD    FLAT:$LN18@main ; inner block filter
                 DD    FLAT:$LN13@main ; inner block handler
xdata$x    ENDS

$T2 = -40         ; size = 4
$T3 = -36         ; size = 4
_p$ = -32         ; size = 4
tv72 = -28        ; size = 4
__$SEHRec$ = -24 ; size = 24
```

```
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2  ; initial previous try level
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax ; prev
    add    esp, -24
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax       ; xored point
er to scope table
    xor    eax, ebp                                ; ebp ^ secur
ity_cookie
    push   eax
    lea    eax, DWORD PTR __$SEHRec$[ebp+8]        ; pointer to
VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try
block, setting previous try level=0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try
block, setting previous try level=1
    push   OFFSET $SG85497 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
    push   0
    push   0
    push   1122867 ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try b
lock, set previous try level back to 0
    jmp    SHORT $LN2@main

; inner block filter:
$LN12@main:
$LN18@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T3[ebp], eax
    cmp    DWORD PTR $T3[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN5@main
```

```
    mov    DWORD PTR tv72[ebp], 1
    jmp    SHORT $LN6@main
$LN5@main:
    mov    DWORD PTR tv72[ebp], 0
$LN6@main:
    mov    eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
    ret    0

; inner block handler:
$LN13@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85501 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try b
lock, setting previous try level back to 0
$LN2@main:
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both block
s, setting previous try level back to -2
    jmp    SHORT $LN7@main

; outer block filter:
$LN8@main:
$LN19@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T2[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$LN10@main:
$LN17@main:
    ret    0

; outer block handler:
$LN9@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85503 ; 'user exception caught'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both block
s, setting previous try level back to -2
$LN7@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
```

```
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret     0
_main    ENDP

_code$ = 8  ; size = 4
_ep$ = 12   ; size = 4
_filter_user_exceptions PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _code$[ebp]
    push    eax
    push    OFFSET $SG85486 ; 'in filter. code=0x%08X'
    call    _printf
    add     esp, 8
    cmp     DWORD PTR _code$[ebp], 1122867 ; 00112233H
    jne     SHORT $LN2@filter_use
    push    OFFSET $SG85488 ; 'yes, that is our exception'
    call    _printf
    add     esp, 4
    mov     eax, 1
    jmp     SHORT $LN3@filter_use
    jmp     SHORT $LN3@filter_use
$LN2@filter_use:
    push    OFFSET $SG85490 ; 'not our exception'
    call    _printf
    add     esp, 4
    xor     eax, eax
$LN3@filter_use:
    pop     ebp
    ret     0
_filter_user_exceptions ENDP
```

这里是cookie的含义：Cookie Offset用于区分栈中saved_EBP的地址和EBP⊕security_cookie。附加的Cookie XOR Offset用于区分EBP⊕security_cookie是否保存在栈中。如果这个等式不为true，会由于栈受到破坏而停止这个过程。

security_cookie⊕(Cookie XOR Offset+address_of_saved_EBP) == stack[address_of_saved_EBP + CookieOffset]

如果Cookie Offset为-2，这意味着它不存在。

在我的tracer工具也实现了Cookie检查，具体请看Github。

MSVC 2005之后的编译器开启/GS选项仍可能会回滚到SEH3。不过，CRT的代码总是使用SEH4。

68.3.3 Windows x64

正如你所认为的，每个函数序言在设置SEH帧效率不高。另一个性能问题是，函数执行期间多次尝试改变previous try level。这种情况在x64完全改变了：现在所有指向try块，filter和handler函数都保存在PE文件的.pdata段，由它提供给操作系统异常处理所需信息。

这里有两个使用x64编译的例子：

Listing 68.12: MSVC 2012

```
$SG86276 DB 'hello #1!', 0aH, 00H
$SG86277 DB 'hello #2!', 0aH, 00H
$SG86279 DB 'access violation, can''t recover', 0aH, 00H
pdata SEGMENT
$pdata$main DD imagerel $LN9
    DD imagerel $LN9+61
    DD imagerel $unwind$main
pdata ENDS
pdata SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
    DD imagerel main$filt$0+32
    DD imagerel $unwind$main$filt$0
pdata ENDS
xdata SEGMENT
$unwind$main DD 020609H
    DD 030023206H
    DD imagerel __C_specific_handler
    DD 01H
    DD imagerel $LN9+8
    DD imagerel $LN9+40
    DD imagerel main$filt$0
    DD imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
    DD 050023206H
xdata ENDS
_TEXT SEGMENT
main PROC
$LN9:
    push rbx
    sub rsp, 32
    xor ebx, ebx
    lea rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
    call printf
    mov DWORD PTR [rbx], 13
    lea rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
    call printf
    jmp SHORT $LN8@main
$LN6@main:
    lea rcx, OFFSET FLAT:$SG86279 ; 'access violation, can''t re
cover'
    call printf
    npad 1 ; align next label
$LN8@main:
```

```
    xor eax, eax
    add rsp, 32
    pop rbx
    ret 0
main ENDP
_TEXT ENDS

text$x SEGMENT
main$filt$0 PROC
    push rbp
    sub rsp, 32
    mov rbp, rdx
$LN5@main$filt$:
    mov rax, QWORD PTR [rcx]
    xor ecx, ecx
    cmp DWORD PTR [rax], -1073741819; c0000005H
    sete cl
    mov eax, ecx
$LN7@main$filt$:
    add rsp, 32
    pop rbp
    ret 0
    int 3
main$filt$0 ENDP
text$x ENDS
```

Listing 68.13: MSVC 2012

```
$SG86277 DB 'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB 'yes, that is our exception', 0aH, 00H
$SG86281 DB 'not our exception', 0aH, 00H
$SG86288 DB 'hello!', 0aH, 00H
$SG86290 DB '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB 'access violation, can''t recover', 0aH, 00H
$SG86294 DB 'user exception caught', 0aH, 00H

pdata SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
    DD imagerel $LN6+73
    DD imagerel $unwind$filter_user_exceptions
$pdata$main DD imagerel $LN14
    DD imagerel $LN14+95
    DD imagerel $unwind$main
pdata ENDS
pdata SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
    DD imagerel main$filt$0+32
    DD imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
    DD imagerel main$filt$1+30
    DD imagerel $unwind$main$filt$1
```

```
pdata ENDS
xdata SEGMENT
$unwind$filter_user_exceptions DD 020601H
    DD 030023206H
$unwind$main DD 020609H
    DD 030023206H
    DD imagerel __C_specific_handler
    DD 02H
    DD imagerel $LN14+8
    DD imagerel $LN14+59
    DD imagerel main$filt$0
    DD imagerel $LN14+59
    DD imagerel $LN14+8
    DD imagerel $LN14+74
    DD imagerel main$filt$1
    DD imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
    DD 050023206H
$unwind$main$filt$1 DD 020601H
    DD 050023206H
xdata ENDS

_TEXT SEGMENT
main PROC
$LN14:
    push rbx
    sub rsp, 32
    xor ebx, ebx
    lea rcx, OFFSET FLAT:$SG86288 ; 'hello!'
    call printf
    xor r9d, r9d
    xor r8d, r8d
    xor edx, edx
    mov ecx, 1122867 ; 00112233H
    call QWORD PTR __imp_RaiseException
    lea rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let''s
 crash'
    call printf
    mov DWORD PTR [rbx], 13
    jmp SHORT $LN13@main
$LN11@main:
    lea rcx, OFFSET FLAT:$SG86292 ; 'access violation, can''t re
cover'
    call printf
    npad 1 ; align next label
$LN13@main:
    jmp SHORT $LN9@main
$LN7@main:
    lea rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
    call printf
    npad 1 ; align next label
$LN9@main:
    xor eax, eax
```

```
    add rsp, 32
    pop rbx
    ret 0
main ENDP

text$x SEGMENT
main$filt$0 PROC
    push rbp
    sub rsp, 32
    mov rbp, rdx
$LN10@main$filt$:
    mov rax, QWORD PTR [rcx]
    xor ecx, ecx
    cmp DWORD PTR [rax], -1073741819; c0000005H
    sete cl
    mov eax, ecx
$LN12@main$filt$:
    add rsp, 32
    pop rbp
    ret 0
    int 3
main$filt$0 ENDP
main$filt$1 PROC
    push rbp
    sub rsp, 32
    mov rbp, rdx
$LN6@main$filt$:
    mov rax, QWORD PTR [rcx]
    mov rdx, rcx
    mov ecx, DWORD PTR [rax]
    call filter_user_exceptions
    npad 1 ; align next label
$LN8@main$filt$:
    add rsp, 32
    pop rbp
    ret 0
    int 3
main$filt$1 ENDP
text$x ENDS

_TEXT SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
    push rbx
    sub rsp, 32
    mov ebx, ecx
    mov edx, ecx
    lea rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x%08X'
    call printf
    cmp ebx, 1122867; 00112233H
    jne SHORT $LN2@filter_use
```

```
    lea rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our exception'
    call printf
    mov eax, 1
    add rsp, 32
    pop rbx
    ret 0
$LN2@filter_use:
    lea rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
    call printf
    xor eax, eax
    add rsp, 32
    pop rbx
    ret 0
filter_user_exceptions ENDP
_TEXT ENDS
```

读Sko12获取更多详细的信息。

除了异常信息，.pdata还包含了几乎所有函数的开始和结束地址，因此它可能对于自动化分析工具有用。

68.3.4 更多关于SEH的信息

Matt Pietrek. “A Crash Course on the Depths of Win32™ Structured Exception Handling”. In: MSDN magazine (). URL: http://go.yurichev.com/17293.

Igor Skochinsky. Compiler Internals: Exceptions and RTTI. Also available as http://go.yurichev.com/ 17294. 2012.

68.4 Windows NT: Critical section

临界区在任何操作系统多线程环境中都是非常重要的，它保证一个线程在某一时刻访问一些数据的时候，阻塞其它正要访问这些数据的线程。

下面是Windows NT操作系统的CRITICAL_SECTION声明：

Listing 68.14: (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;
    //
    // The following three fields control entering and exiting t
he critical
    // section for the resource
    //
    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread; // from the thread's ClientId->UniqueTh
read
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount; // force size on 64-bit systems when pa
cked
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

下面展示了EnterCriticalSection()函数的运行过程：

Listing 68.15: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlEnterCriticalSection@4
var_C = dword ptr -0Ch
var_8 = dword ptr -8
var_4 = dword ptr -4
arg_0 = dword ptr 8
    mov edi, edi
    push ebp
    mov ebp, esp
    sub esp, 0Ch
    push esi
    push edi
    mov edi, [ebp+arg_0]
    lea esi, [edi+4] ; LockCount
    mov eax, esi
    lock btr dword ptr [eax], 0
    jnb wait ; jump if CF=0
loc_7DE922DD:
    mov eax, large fs:18h
    mov ecx, [eax+24h]
    mov [edi+0Ch], ecx
    mov dword ptr [edi+8], 1
    pop edi
    xor eax, eax
    pop esi
    mov esp, ebp
    pop ebp
    retn 4
... skipped
```

在这段代码中最重要的指令是BTR(带LOCK前缀)：把目的操作数中由源操作数所指定位的值送往标志位CF，并将目的操作数中的该位置0。这是一个原子操作，会阻塞掉其它同时想要访问这段内存的CPU（参看BTR指令的LOCK前缀）。如果LockCount是1，则重置并返回：我们现在正处于临界区。如果不是，则表示其它线程正在占用，将进入等待状态。

使用WaitForSingleObject()进入等待状态。

下面展示了LeaveCriticalSection()函数的运行过程：

Listing 68.16: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlLeaveCriticalSection@4 proc near
arg_0 = dword ptr 8
    mov edi, edi
    push ebp
    mov ebp, esp
    push esi
    mov esi, [ebp+arg_0]
    add dword ptr [esi+8], 0FFFFFFFFh ;RecursionCount
    jnz short loc_7DE922B2
    push ebx
    push edi
    lea edi, [esi+4] ; LockCount
    mov dword ptr [esi+0Ch], 0
    mov ebx, 1
    mov eax, edi
    lock xadd [eax], ebx
    inc ebx
    cmp ebx, 0FFFFFFFFh
    jnz loc_7DEA8EB7
    loc_7DE922B0:
    pop edi
    pop ebx
loc_7DE922B2:
    xor eax, eax
    pop esi
    pop ebp
    retn 4
... skipped
```

XADD指令功能是：交换并相加。这种情况下，LockCount加1并把结果保存到EBX寄存器，同时把1赋值给LockCount。这个操作是原子的，因为它使用了LOCK前缀，这意味着系统会阻塞其它CPU或CPU核心同时访问这块内存。

LOCK前缀是非常重要的：如果两个线程，每个都工作在不同的CPU或CPU核心，它们都能够进入critical section并修改内存数据，这种行为将导致不确定的后果。

Part VII 工具

第六十九章

反汇编器

69.1 IDA

较老的可下载的免费版本:http://go.yurichev.com/17031

短热键列表(第977页)

第七十章

调试器

70.1 OllyDbg

非常流行的win32用户态调试器 http://go.yurichev.com/17032 短热键列表(第977页)

70.2 GDB

GDB在逆向工程师中并不非常流行，但用起来非常舒适。部分命令（第978页）

70.3 tracer

我用tracer代替调试器。

我最终不再使用调试器是因为我所需要的只是在代码执行的时候找到函数的参数，或者寄存器在某点的状态。每次加载调试器的时间太长，因此我编写了一个小工具tracer。它有控制台接口，运行在命令行下，允许我们给函数下断，查看寄存器状态，修改值等等。

但出于学习的目的更建议在调试器中手动跟踪代码，观察寄存器状态是怎么变化的(比如经典的SoftICE,Ollydbg,Windbg寄存器值发生变化会高亮)，手动修改标志位，数据然后观察效果。

第七十一章

系统调用跟踪

71.0.1 stace/dtruss

显示当前进程的系统调用(第697页)。比如：

```
# strace df -h
...
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such fil
e or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232
\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWR
ITE, 3, 0) = 0xb75b3000
```

Mac OS X 的dtruss也有这个功能。

Cygwin也有strace，但如果我理解正确的话，它只为cygwin环境编译exe文件工作。

第七十二章

反编译器

只有一个已知的，公开的，高质量的反编译C代码的反编译器：Hex-Rays

http://go.yurichev.com/17033

第七十三章

其他工具

- Microsoft Visual Studio Express1:Visual Studio精简版，方便做简单的实验。部分有用的选项(第978页)
- Hiew：适用于二进制文件小型修改
- binary grep:大量文件中搜索常量(或者任何有序字节)的小工具，也可以用于不可执行文件：GitHub

Part IX 逆向文件格式的例子

第八十四章

简单异或加密

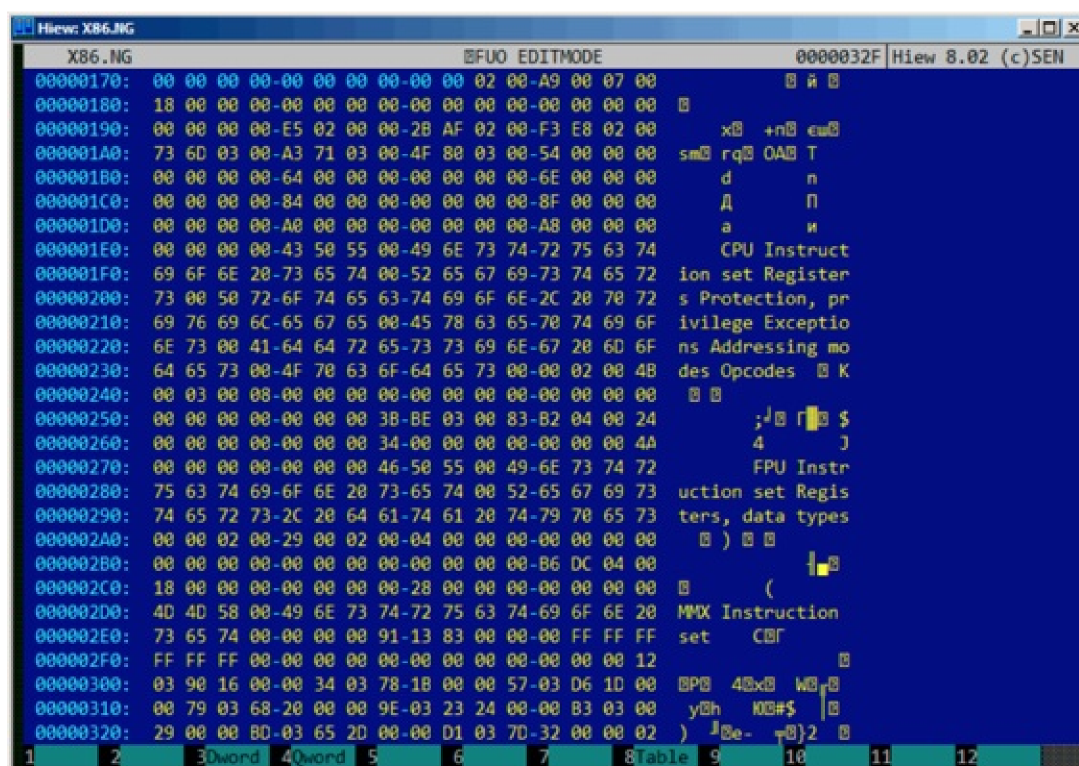
84.1

Norton Guide这款工具在MS-DOS时代很受欢迎，作为超文本参考手册程序常驻在系统中。 Norton Guide的数据库文件扩展名是.ng，内容看上去是加密的：

```
view X86.NG - Far 2.0.1807 x64 Administrator
U:\retrocomputing\MS-DOS\norton guide\X86.NG            866        372131      Col 0        0%
0000000170: 00 00 00 00 00 00 00 00  00 00 18 1A B3 1A 1D 1A            ↑→│↔→
0000000180: 02 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ☻→→→→→→→→→→→→→→→
0000000190: 1A 1A 1A 1A FF 18 1A 1A  31 B5 18 1A E9 F2 18 1A  →→→→ ↑→→1╡↑→щЄ↑→
00000001A0: 69 77 19 1A B9 6B 19 1A  55 9A 19 1A 4E 1A 1A 1A  iw↓→╣k↓→UЪ↓→N→→→
00000001B0: 1A 1A 1A 1A 7E 1A 1A 1A  1A 1A 1A 1A 74 1A 1A 1A  →→→→~→→→→→→→t→→→
00000001C0: 1A 1A 1A 1A 9E 1A 1A 1A  1A 1A 1A 1A 95 1A 1A 1A  →→→→Ю→→→→→→→Х→→→
00000001D0: 1A 1A 1A 1A BA 1A 1A 1A  1A 1A 1A 1A B2 1A 1A 1A  →→→→║→→→→→→→▓→→→
00000001E0: 1A 1A 1A 1A 59 4A 4F 1A  53 74 69 6E 68 6F 79 6E  →→→→YJO→Stinhoyn
00000001F0: 73 75 74 3A 69 7F 6E 1A  48 7F 7D 73 69 6E 7F 68  sut:i⌂n→H⌂}sin⌂h
0000000200: 69 1A 4A 68 75 6E 7F 79  6E 73 75 74 36 3A 6A 68  i→Jhun⌂ynsut6:jh
0000000210: 73 6C 73 76 7F 7D 7F 1A  5F 62 79 7F 6A 6E 73 75  slsv⌂}⌂→_by⌂jnsu
0000000220: 74 69 1A 5B 7E 7E 68 7F  69 69 73 74 7D 3A 77 75  ti→[~~h⌂iist}:wu
0000000230: 7E 7F 69 1A 55 6A 79 75  7E 7F 69 1A 1A 18 1A 51  ~⌂i→Ujyu~⌂i→→↑→Q
0000000240: 1A 19 1A 12 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  →↓→↕→→→→→→→→→→→→
0000000250: 1A 1A 1A 1A 1A 1A 1A 21  A4 19 1A 99 A8 1E 1A 3E  →→→→→→→!д↓→Щи▲→>
0000000260: 1A 1A 1A 1A 1A 1A 1A 2E  1A 1A 1A 1A 1A 1A 1A 50  →→→→→→→.→→→→→→→P
0000000270: 1A 1A 1A 1A 1A 1A 1A 5C  4A 4F 1A 53 74 69 6E 68  →→→→→→→\JO→Stinh
0000000280: 6F 79 6E 73 75 74 3A 69  7F 6E 1A 48 7F 7D 73 69  oynsut:i⌂n→H⌂}si
0000000290: 6E 7F 68 69 36 3A 7E 7B  6E 7B 3A 6E 63 6A 7F 69  n⌂hi6:~{n{:ncj⌂i
00000002A0: 1A 1A 18 1A 33 1A 18 1A  1E 1A 1A 1A 1A 1A 1A 1A  →→↑→3→↑→▲→→→→→→→
00000002B0: 1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A AC C6 1E 1A  →→→→→→→→→→→→м╞▲→
00000002C0: 02 1A 1A 1A 1A 1A 1A 1A  32 1A 1A 1A 1A 1A 1A 1A  ☻→→→→→→→2→→→→→→→
00000002D0: 57 57 42 1A 53 74 69 6E  68 6F 79 6E 73 75 74 3A  WWB→Stinhoynsut:
00000002E0: 69 7F 6E 1A 1A 1A 1A 8B  09 99 1A 1A 1A E5 E5 E5  i⌂n→→→→Л○Щ→→→xxx
00000002F0: E5 E5 E5 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 08  xxx→→→→→→→→→→→→◘
0000000300: 19 8A 0C 1A 1A 2E 19 62  01 1A 1A 4D 19 CC 07 1A  ↓K♀→→.↓b☺→→M↓╠•→
0000000310: 1A 63 19 72 3A 1A 1A 84  19 39 3E 1A 1A A9 19 1A  →c↓r:→→Д↓9>→→й↓→
0000000320: 33 1A 1A A7 19 7F 37 1A  1A CB 19 67 28 1A 1A 18  3→→з↓⌂7→→╦↓g(→→↑
1      2      3      4      5Print 6      7Prev  8Goto  9Video 10     11ViewHs 12
```

为什么我说内容是加密的而不是压缩的呢？可以看到，0x1A字节(看起来是“→”)经常出现，而在压缩文件中不会有这种情况,所以这是个加密文件。同时我们也发现大段只包含拉丁字母的部分，看上去就像未知语言的字符串。

0x1A字节出现得频率很高，我们可以尝试解密这个文件，先假设它是用最简单的异或加密。如果我们用0x1A和Hiew中的每个字节异或，我们就能看见熟悉的英文字符串：



与单个固定字节异或是最简单的可能的加密方法，有时可能会碰到。

现在我们理解了为什么0x1A出现的频率如此高了：因为文件包含了大量0字节，加密之后替换成了0x1A。

但是常量可能会不同。在这个例子中，我们可以尝试0到255之间的每一个常量，在解密文件中寻找熟悉的内容，256就不行了。

更多关于Norton Guide文件格式内容： http://go.yurichev.com/17317

84.1.1 熵

像这样简单的加密系统一个很重要的特性就是加密/解密块的信息熵是一样的。下面是我用 Wolfram Mathematica 10的分析。

```
In[1]:= input = BinaryReadList["X86.NG"];
In[2]:= Entropy[2, input] // N
Out[2]= 5.62724
In[3]:= decrypted = Map[BitXor[#, 16^^1A] &, input];
In[4]:= Export["X86_decrypted.NG", decrypted, "Binary"];

In[5]:= Entropy[2, decrypted] // N
Out[5]= 5.62724
In[6]:= Entropy[2, ExampleData[{"Text", "ShakespearesSonnets"}]]
 // N
Out[6]= 4.42366
```

我所做是加载文件，获取它的熵，解密保存之后再次获取它的熵(竟然是一样的！)。Mathematica也提供一些著名了英文文本来分析。所以我获取了莎士比亚十四行诗的熵，它很接近我们所分析的文件。我们分析文件包含了英文句子，和莎士比亚十四行诗的语言接近。使用了异或的英文文本有相同的熵。

然而当文件使用大于一个字节来异或时就不可信了。

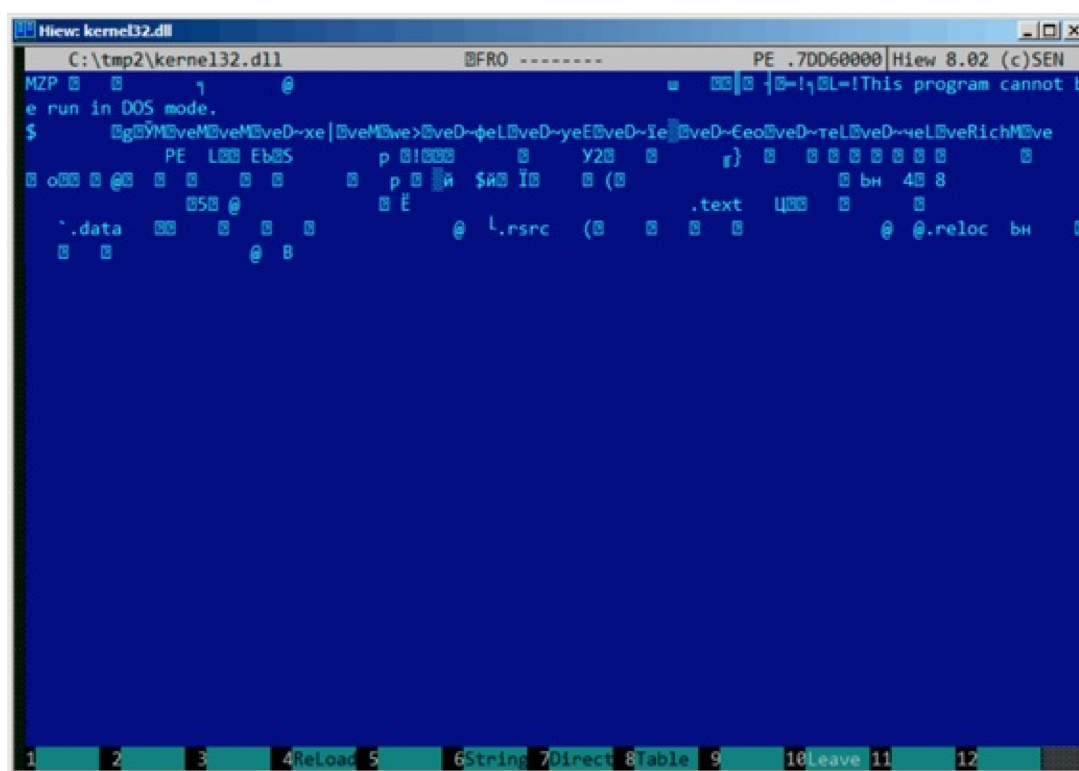
我们分析的文件可以在这里下载到http://go.yurichev.com/17350

关于熵的基数多说一点

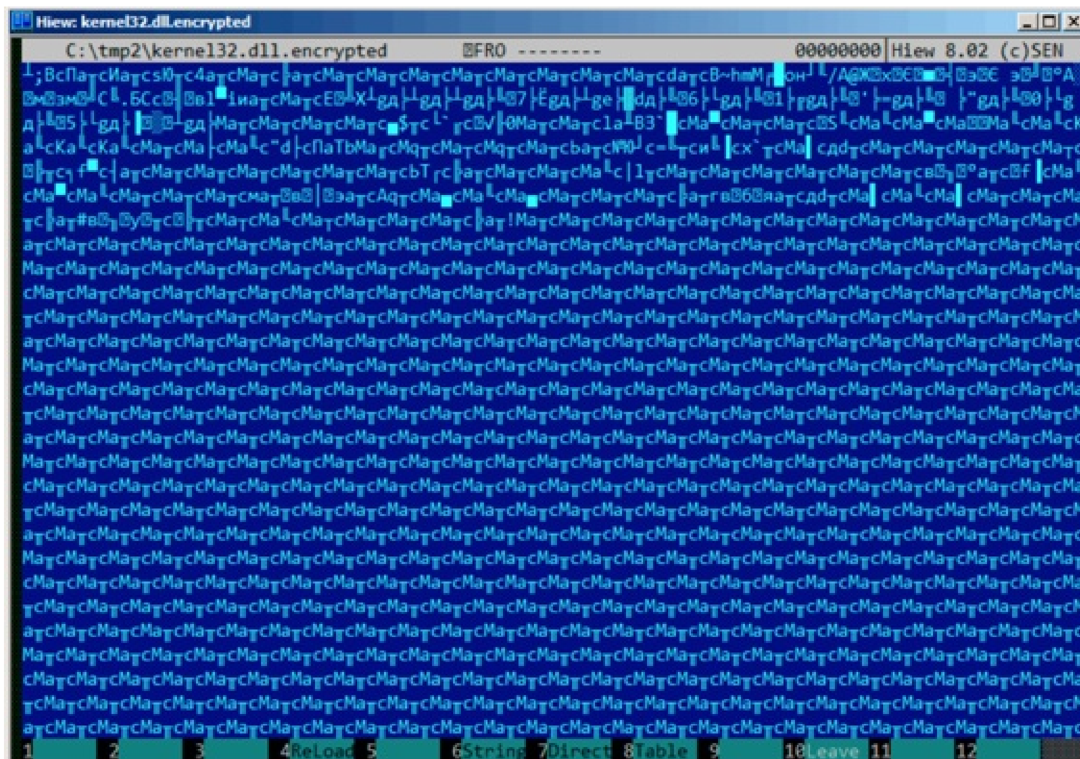
Wolfram Mathematica使用e(自然对数)为基数计算，UNIX的ent工具使用2为基数。所以我在熵命令中将2设为基数，所以Mathematica获得的结果和ent一样。

84.2 最简单4字节异或加密

如果异或加密的时候使用了更长的模式，比如，4字节模式，那么也很容易发现。下面这个例子是kernel32.dll文件的起始部分(Windows Server 2008 32位版本)：



下面是使用4字节密钥“加密”的结果：



容易发现有四个字符重复出现。事实上，PE文件头有许多0字节填充区，这也是密钥能被看出来的原因。

下面是十六进制形式PE头的开头：



下面是“加密”后：



容易发现密钥是这四个字节：8C 61 D3 63 . 使用这个信息解密整个文件很容易。因此记住PE文件的这些特性是很重要的：1）PE头有许多0字节填充区；2）每页有4096字节，所有的PE区段用0补齐，经常可以看到所有的区段后出现很长的0字节填充区。

一些其他的文件格式可能包含长0字节填充区，对于科学和工程软件文件来说非常典型。

想自己分析这些文件可以到这里下载：http://go.yurichev.com/

84.2.1 练习

作为一个练习尝试解密下面这个文件。当然，密钥已经改变。http://go.yurichev.com/17353

第八十五章

Millenium 游戏存档文件

"Millenium Return to Earth"是一款很老的DOS游戏(1991)，你可以挖掘矿产资源，修建船只，在其他星球上装备它们等等。

就像其他游戏一样，你可以将游戏状态存入一个文件中。

咱们来看看能不能找到点什么。

下面这是游戏中的一处矿井。有些星球的矿井工作更快，也有工作慢的。资源的设置也不同。来看看现在底下埋的是什么样的资源：



我保存了游戏状态。存档文件大小为9538字节。

我在游戏中等了"几天"，现在我们可以从矿井中得到更多的资源：



我再次保存了游戏状态。

下面我们使用简单的DOS/Windows FC 工具来比较二进制存档文件。

```
...> FC /b 2200save.i.v1 2200SAVE.I.V2

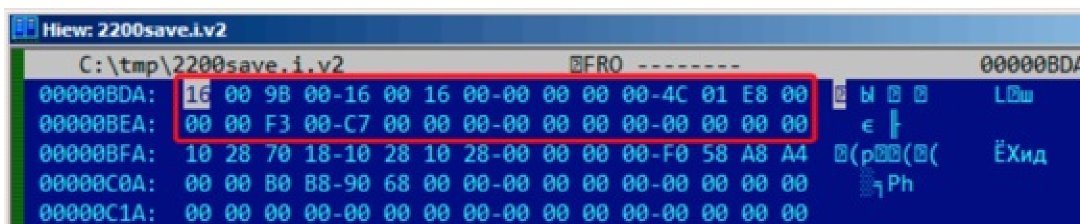
Comparing files 2200save.i.v1 and 2200SAVE.I.V2
00000016: 0D 04
00000017: 03 04
0000001C: 1F 1E
00000146: 27 3B
00000BDA: 0E 16
00000BDC: 66 9B
00000BDE: 0E 16
00000BE0: 0E 16
00000BE6: DB 4C
00000BE7: 00 01
00000BE8: 99 E8
00000BEC: A1 F3
00000BEE: 83 C7
00000BFB: A8 28
00000BFD: 98 18
00000BFF: A8 28
00000C01: A8 28
00000C07: D8 58
00000C09: E4 A4
00000C0D: 38 B8
00000C0F: E8 68

...
```

这里的输出并不完整，有许多不同之处，我截取了结果来展示最有趣的部分。

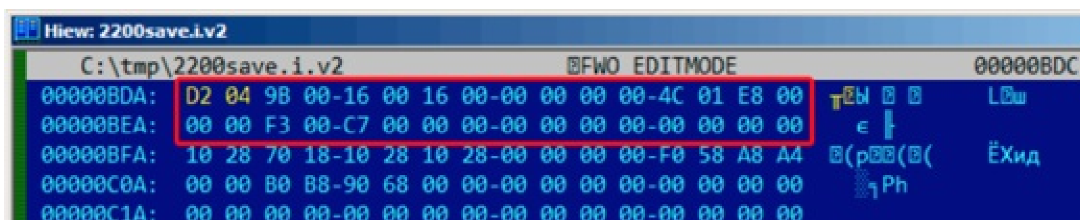
在第一个状态中，我有14个单位的氢气，102个单位的氧气。在第二个状态中则分别是22和155个单位。如果这些值保存到了存档文件中，我们会看到差异。的确如此，较老存档的0XBDA处的值为0x0E(14),而在新存档中则变成了0x16(22)。这可能是氢气。同样，老存档的0XBDC处的值为0x66(102)，新存档中值变为0x9B(155)。这看上去是氧气。我将两个文件都放在我的网站上，想要自己实验了解更多信息的请戳这里：beginners.re

下面是在Hiew中显示的新存档文件，我将游戏中与资源相关的值标记了出来：



我检查后确认它们是16-bit值，不是16-bit DOS软件中什么奇怪的东西，16-bit的DOS软件int类型为16比特。

下面来验证咱们的假设吧。我在第一个位置(氢气的位置)写入1234(0x4D2):

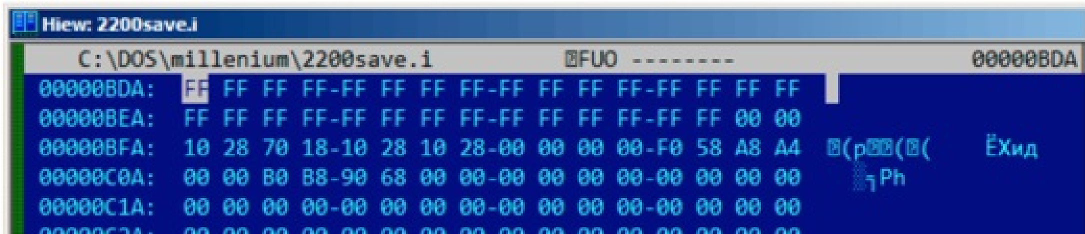


然后我加载这个已改变的文件到游戏中，查看矿井的数据：



这就对了。

现在我们尝试让这个游戏更快结束，把值设为最大：



0xFFFF就是65536，现在我们拥有了许多资源：



我在游戏中跳过了"几天"，结果有些资源变少了：



溢出发生了。游戏开发者可能没考虑到会出现这么多的资源的情况，没有设置溢出检查，但游戏中的矿井仍然在工作，资源在增加，所以导致了溢出。我想我不应该那么贪婪。

可能大量的数值都保存在了这个文件中。

这就是非常简单的游戏欺骗方法。高分文件可以通过这样打补丁轻松得到。

更多关于文件和内存快照的比对：63.4 第681页

第八十六章

Oracle RDBMS: .SYM-files

当一个Oracle RDBMS进程出于某种原因崩溃时，会将许多信息写入日志文件，包括栈回溯，就像这样：

```
----- Call Stack Trace -----
calling              call     entry                         argument
 values in hex
location             type     point                      (? means du
bious value)
-------------------- -------- --------------------     ----------
------------------
_kqvrow()                      00000000
_opifch2()+2729         CALLptr 00000000                    23D4B91
4 E47F264 1F19AE2
                                                     EB1C8A8 1
_kpoal8()+2832         CALLrel _opifch2()                   89 5 EB1
CC74
_opiodr()+1248         CALLreg 00000000                  5E 1C EB1F
0A0
_ttcpip()+1051         CALLreg 00000000                  5E 1C EB1F
0A0 0
_opitsk()+1404         CALL??? 00000000                  C96C040 5E
 EB1F0A0 0 EB1ED30
                                                     EB1F1CC 53E5
2E 0 EB1F1F8
_opiino()+980         CALLrel    _opitsk()                  0   0
_opiodr()+1248         CALLreg    00000000                  3C 4 EB
1FBF4
_opidrv()+1201         CALLrel    _opiodr()                  3C 4 E
B1FBF4 0
_sou2o()+55             CALLrel    _opidrv()                  3C 4
EB1FBF4
_opimai_real()+124     CALLrel    _opimai_real()               2 EB1
FC2C
_OracleThreadStart@ CALLrel    _opimai()                  2 EB1FF6
C 7C88A7F4 EB1FC34 0
4()+830                                                     EB1FD04
77E6481C             CALLreg 00000000                E41FF9C 0 0
E41FF9C 0 EB1FFC4
00000000             CALL??? 00000000
```

当然Oracle RDBMS 可执行文件肯定拥有某种调试信息,或者带有符号信息(或者类似信息)的映射文件。

WindowsNT Oracle RDBMS 的符号信息包含在具有.SYM扩展名的文件中，具有专门的格式。(平文文件挺好的，但是需要额外解析，因此获取速度更慢)

咱们来看看能不能理解它的格式。我选了最短的orawtc8.sym文件，来自于Oracle 8.1.7 版本orawtc8.dll 文件。

下面是Hiew加载后的效果：

通过与其他.SYN文件的对比，我们可以快速发现"OSYM"总是头(和尾)，因此这可能就是文件的标志。

同时也可以看出，文件格式是：OSYM+一些二进制数据+以0为界定符的字符串+OSYM。这些字符串显然是函数和全局变量名。

我标记了OSYM标志和字符串：

咱们来看看。我在Hiew中标记了整个字符串块(除了末尾的OSYM)，然后把它放进单独的文件中。然后我运行UNIX的strings和wc工具分析字符串

```
strings strings_block | wc -l
66
```

有66个文本字符串，请记住这个数字。

可以这么说，常规情况下，数量值会被存储在一个单独的二进制文件中。的确也是如此，我们可以在文件开头找到这个66数字(0x42)，就在OSYM这个标志右边：

```
$ hexdump -C orawtc8.sym
00000000  4f 53 59 4d 42 00 00 00  00 10 00 10 80 10 00 10  |OSY
MB..........|
00000010  f0 10 00 10 50 11 00 10  60 11 00 10 c0 11 00 10  |...
.P...`.......|
00000020  d0 11 00 10 70 13 00 10  40 15 00 10 50 15 00 10  |...
.p...@...P...|
00000030  60 15 00 10 80 15 00 10  a0 15 00 10 a6 15 00 10  |`..
.............|
....
```

当然，这里的0x42不是一个字节，更像是一个32比特的值，小端，后面至少跟着3个0字节。

为什么我认为是32位呢？因为Oracle RDBMS的符号文件比较大。主要的oracle.exe可执行文件(10.2.0.4版本)的oracle.sym文件包含0x3A38E(238478)个符号。一个16位的值在这里明显不够。

我检查了其他.SYM文件，证实了我的猜想：OSYM符号后面的32位值总表示文件字符串的数量。

这对于所有的二进制文件来说几乎是通用的：文件头包含标志和文件其他信息。

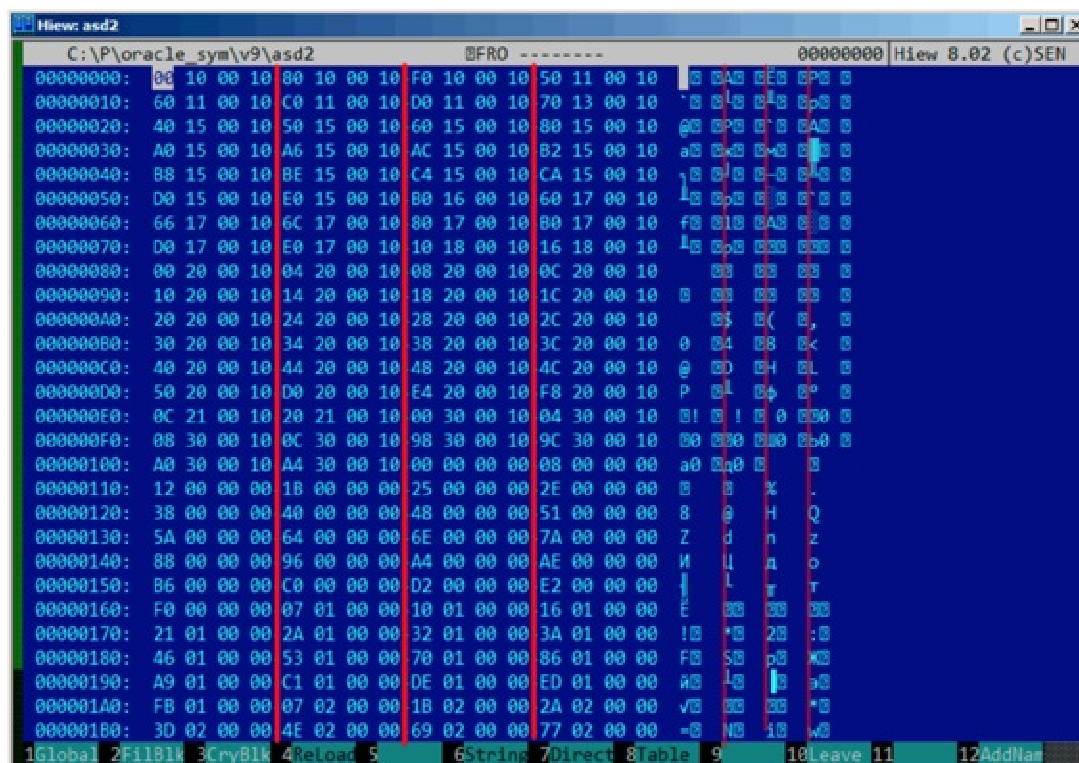
现在我们来进一步调查二进制块是什么。再次使用Hiew，我把从块头8个字节(32位计数值后面)开始一直到字符串块结尾的内容放入单独的文件中。

在Hiew中看看这个二进制块：



有一个明显的规律。

我用红线划分了这个块：



Hiew,就像其他的十六进制编辑器一样，每行显示16个字节。所以规律很容易看出来：每行有4个32位的值。

这个规律容易看出来的原因是其中的一些值(地址0x104之前)总是具有0x1000xxxx的格式，以0x10和0字节开始。其他值(从地址0x108开始)都是0x0000xxxx的格式，总是以两个0字节开始。

我们把这个块当作32位值的数组dump出来：

```
$ od -v -t x4 binary_block
0000000 10001000 10001080 100010f0 10001150
0000020 10001160 100011c0 100011d0 10001370
0000040 10001540 10001550 10001560 10001580
0000060 100015a0 100015a6 100015ac 100015b2
0000100 100015b8 100015be 100015c4 100015ca
0000120 100015d0 100015e0 100016b0 10001760
0000140 10001766 1000176c 10001780 100017b0
0000160 100017d0 100017e0 10001810 10001816
0000200 10002000 10002004 10002008 1000200c
0000220 10002010 10002014 10002018 1000201c
0000240 10002020 10002024 10002028 1000202c
0000260 10002030 10002034 10002038 1000203c
0000300 10002040 10002044 10002048 1000204c
0000320 10002050 100020d0 100020e4 100020f8
0000340 1000210c 10002120 10003000 10003004
0000360 10003008 1000300c 10003098 1000309c
0000400 100030a0 100030a4 00000000 00000008
0000420 00000012 0000001b 00000025 0000002e
0000440 00000038 00000040 00000048 00000051

0000460 0000005a 00000064 0000006e 0000007a
0000500 00000088 00000096 000000a4 000000ae
0000520 000000b6 000000c0 000000d2 000000e2
0000540 000000f0 00000107 00000110 00000116
0000560 00000121 0000012a 00000132 0000013a
0000600 00000146 00000153 00000170 00000186
0000620 000001a9 000001c1 000001de 000001ed
0000640 000001fb 00000207 0000021b 0000022a
0000660 0000023d 0000024e 00000269 00000277
0000700 00000287 00000297 000002b6 000002ca
0000720 000002dc 000002f0 00000304 00000321
0000740 0000033e 0000035d 0000037a 00000395
0000760 000003ae 000003b6 000003be 000003c6
0001000 000003ce 000003dc 000003e9 000003f8
0001020
```

这里有132个值，也就是66*2。或许每一个符号有两个32位的值，或者有两个数组呢？咱们接着看。

从0x1000开始的值可能是地址。毕竟这是dll的.SYM文件，win32 DLL默认的基址是0x10000000，代码通常从0x10001000开始。

我用IDA打开orawtc8.dll文件时发现基址并不相同，不过没关系，第一个函数是：

```
.text:60351000 sub_60351000    proc near
.text:60351000
.text:60351000 arg_0        = dword ptr 8
.text:60351000 arg_4        = dword ptr 0Ch
.text:60351000 arg_8        = dword ptr 10Ch
.text:60351000
.text:60351000                 push     ebp
.text:60351001                mov        ebp,esp
.text:60351003                mov        eax, dword_60353014
.text:60351008                cmp        eax, 0FFFFFFFFh
.text:6035100B                jnz        short loc_6035104F
.text:6035100D                mov        ecx, hModule
.text:60351013                xor        eax, eax
.text:60351015                cmp        ecx, 0FFFFFFFFh
.text:60351018                mov        dword_60353014, eax
.text:6035101D                jnz        short loc_60351031
.text:6035101F                call    sub_603510F0
.text:60351024                mov        ecx, eax
.text:60351026                mov        eax, dword_60353014
.text:6035102B                mov        hModule, ecx
.text:60351031
.text:60351031 loc_60351031:                    ; CODE XREF: sub
_60351000+1D
.text:60351031                 test    ecx, ecx
.text:60351033                jbe        short loc_6035104F
.text:60351035                push    offset ProcName ; "ax_reg"
.text:6035103A                push    ecx             ; hModule
.text:6035103B                call    ds:GetProcAddress
...
```

哇，"ax_reg"字符串看起来很熟悉。它的确是字符串块的第一个字符串。所以函数名是"ax_reg"。

第二个函数是：

```
.text:60351080 sub_60351080        proc near
.text:60351080
.text:60351080 arg_0             = dword ptr  8
.text:60351080 arg_4             = dword ptr  0Ch
.text:60351080
.text:60351080                    push    ebp
.text:60351081                    mov     ebp, esp
.text:60351083                    mov     eax, dword_60353018
.text:60351088                    cmp     eax, 0FFFFFFFFh
.text:6035108B                    jnz     short loc_603510CF
.text:6035108D                    mov     ecx, hModule
.text:60351093                    xor     eax, eax
.text:60351095                    cmp     ecx, 0FFFFFFFFh
.text:60351098                    mov     dword_60353018, eax
.text:6035109D                    jnz     short loc_603510B1
.text:6035109F                    call    sub_603510F0
.text:603510A4                    mov     ecx, eax
.text:603510A6                    mov     eax, dword_60353018
.text:603510AB                  mov     hModule, ecx
.text:603510B1
.text:603510B1 loc_603510B1:                             ; CODE
XREF: sub_60351080+1D
.text:603510B1                     test    ecx, ecx
.text:603510B3                    jbe     short loc_603510CF
.text:603510B5                    push    offset aAx_unreg ; "ax
_unreg"
.text:603510BA                     push    ecx              ; hM
odule
.text:603510BB                    call    ds:GetProcAddress
...
```

"ax_unreg"字符串也是字符串块的第二个字符串！第二个函数的开始地址是0x60351080，二进制块的第二个值是10001080.因此就是这个地址，但对于DLL加上了默认基地址

现在可以快速的检查然后确定数组开始的66个值(也就是数组前一半)只是DLL中的函数地址，包括一些标签等等。那么另一半是什么呢？剩余的66个值都是以0x0000开始的，看上去范围是[0...0x3FB8]。并且他们看上去不像位域：序列的数量在增长。最后一个十六进制数字看上去是随机的。因此不像是地址(如果它是4字节，8字节，16字节则可除尽)

我们问问自己吧：Oracle RDBMS的开发者还会在文件中保存什么呢？随便猜猜：可能是文本字符串(函数名)的地址。可以迅速验证这一点，是的，每个数字代表的就是字符串在这个块中第一个字符的位置。

就是这样，完成了！

我写了一个工具将这些.SYM文件转换到IDA脚本中，然后我可以加载.IDC脚本，设置函数名：

```
#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include <assert.h>
#include <malloc.h>
#include <fcntl.h>
#include <string.h>
int main (int argc, char *argv[])
{
        uint32_t sig, cnt, offset;
        uint32_t *d1, *d2;
        int     h, i, remain, file_len;
        char    *d3;
        uint32_t array_size_in_bytes;
        assert (argv[1]); // file name
        assert (argv[2]); // additional offset (if needed)
        // additional offset
        assert (sscanf (argv[2], "%X", &offset)==1);
        // get file length
        assert ((h=open (argv[1], _O_RDONLY | _O_BINARY, 0))!=-1
);
        assert ((file_len=lseek (h, 0, SEEK_END))!=-1);
        assert (lseek (h, 0, SEEK_SET)!=-1);
        // read signature
        assert (read (h, &sig, 4)==4);
        // read count
        assert (read (h, &cnt, 4)==4);
        assert (sig==0x4D59534F); // OSYM
        // skip timedatestamp (for 11g)
        //_lseek (h, 4, 1);
        array_size_in_bytes=cnt*sizeof(uint32_t);
        // load symbol addresses array
        d1=(uint32_t*)malloc (array_size_in_bytes);
        assert (d1);
        assert (read (h, d1, array_size_in_bytes)==array_size_in
_bytes);
        // load string offsets array
        d2=(uint32_t*)malloc (array_size_in_bytes);
        assert (d2);
        assert (read (h, d2, array_size_in_bytes)==array_size_in
_bytes);
        // calculate strings block size
        remain=file_len-(8+4)-(cnt*8);
        // load strings block
        assert (d3=(char*)malloc (remain));
        assert (read (h, d3, remain)==remain);
        printf ("#include <idc.idc>\n\n");
        printf ("static main() {\n");
        for (i=0; i<cnt; i++)
                printf ("\tMakeName(0x%08X, \"%s\");\n", offset
+ d1[i], &d3[d2[i]]);
        printf ("}\n");
```

```
        close (h);
        free (d1); free (d2); free (d3);
```

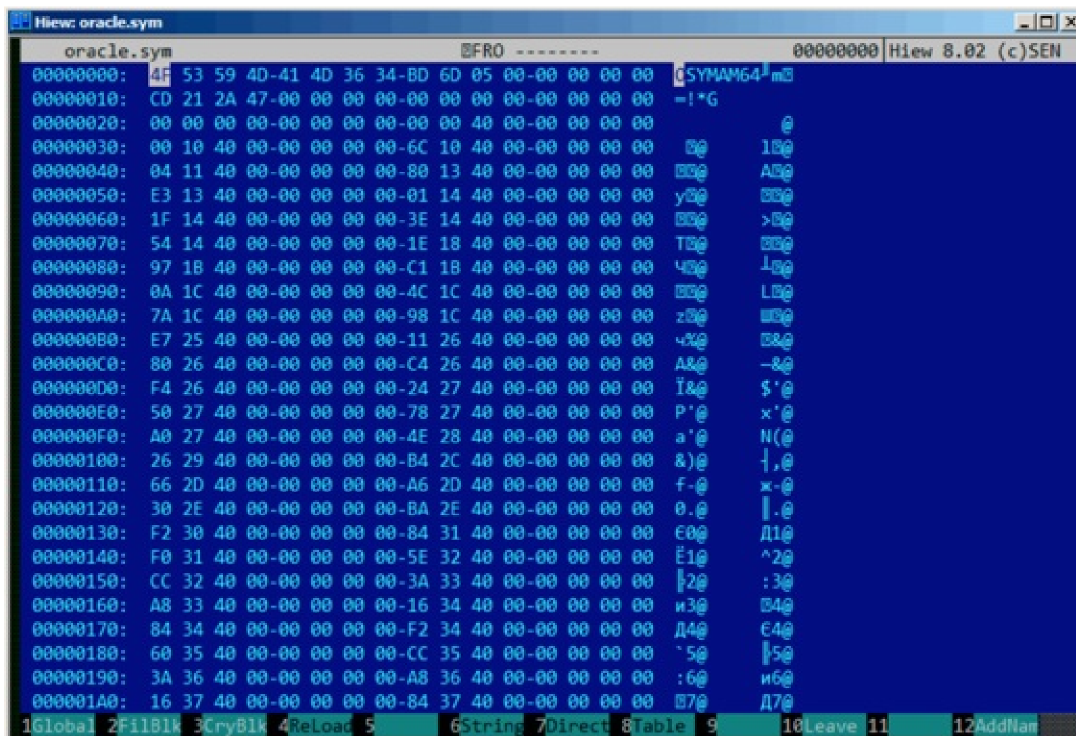
下面是它工作的一个例子：

```
#include <idc.idc>
static main() {
    MakeName(0x60351000, "_ax_reg");
    MakeName(0x60351080, "_ax_unreg");
    MakeName(0x603510F0, "_loaddll");
    MakeName(0x60351150, "_wtcsrin0");
    MakeName(0x60351160, "_wtcsrin");
    MakeName(0x603511C0, "_wtcsrfre");
    MakeName(0x603511D0, "_wtclkm");
    MakeName(0x60351370, "_wtcstu");
... }
```

我使用的例子可以在这里找到：beginners.re

咱们来试试64位的Oracle RDBMS。相应的，地址应该为64位，对么？

8字节的规律看上去更加明显了：



是的，所有的表含有64位的元素，甚至是字符串的偏移。现在标志也变成了OSYMAM64，猜测是用于区分目标平台的。

就是这样了。连接Oracle RDBMS-SYM文件我用的函数的库：GitHub

第八十七章

Oracle RDBMS:.MSB-files

这个二进制文件包含了错误信息和对应的错误码。我们来理解它的格式然后找到unpack方法。

这里有文本格式的Oracle RDBMS错误信息，因此我们可以比对文本和pack后的二进制文件。

下面是ORAUS.MSG文本文件的开头，一些无关紧要的注释已经去掉：

```
00000, 00000, "normal, successful completion"
00001, 00000, "unique constraint (%s.%s) violated"
00017, 00000, "session requested to set trace event"
00018, 00000, "maximum number of sessions exceeded"
00019, 00000, "maximum number of session licenses exceeded"
00020, 00000, "maximum number of processes (%s) exceeded"
00021, 00000, "session attached to some other process; cannot sw
itch session"
00022, 00000, "invalid session ID; access denied"
00023, 00000, "session references process private memory; cannot
 detach session"
00024, 00000, "logins from more than one process not allowed in
single-process mode"
00025, 00000, "failed to allocate %s"
00026, 00000, "missing or invalid session ID"
00027, 00000, "cannot kill current session"
00028, 00000, "your session has been killed"
00029, 00000, "session is not a user session"
00030, 00000, "User session ID does not exist."
00031, 00000, "session marked for kill"
...
```

第一个数字是错误码，第二个可能是附加的标志，我不太确定。

现在我们来打开ORAUS.MSB二进制文件，找到这些文本字符串。这里有：

可以看到，这些文本字符串之间(包括ORAUS.MSG文件开头的那些)插入了一些二进制值。通过快速调查分析可发现二进制文件的主要部分按0x200(512)字节的大小进行分割。

咱们来看看第一个块的内容：

这里可以看到第一条错误信息文本。同时也看到错误信息之间没有0字节。这意味着没有以null结尾的c字符串。因此，每一条错误信息的长度值肯定以某种形式加密了。我们再来找找错误码。ORAUS.MSG文件这样开始：0，1，17(0x11),18

(0x12), 19 (0x13), 20 (0x14), 21 (0x15), 22 (0x16), 23 (0x17), 24 (0x18)...我在块头找到这些数字并且用红线标注出来了。错误码的间隔是6个字节。这意味着可能有6个字节分配给每条错误信息。

第一个16位值(0xA或者10)表示每个块的消息数量：我通过分析其他块证实了这一点，的确是这样：错误信息大小不定，有的长有的短。但是块大小总是固定的，所以你永远也不知道每个块可以pack多少条文本信息。

我注意到，既然这些c字符串不以null结尾，那么他们的大小一定在某处被加密了。第一个字符串"normal, successful completion"的大小是29(0x1D)字节。第二个字符串"unique constraint (%s.%s) violated"的大小是34(0x22)字节。在块里面找不到这些值(0x1D和0x22)。

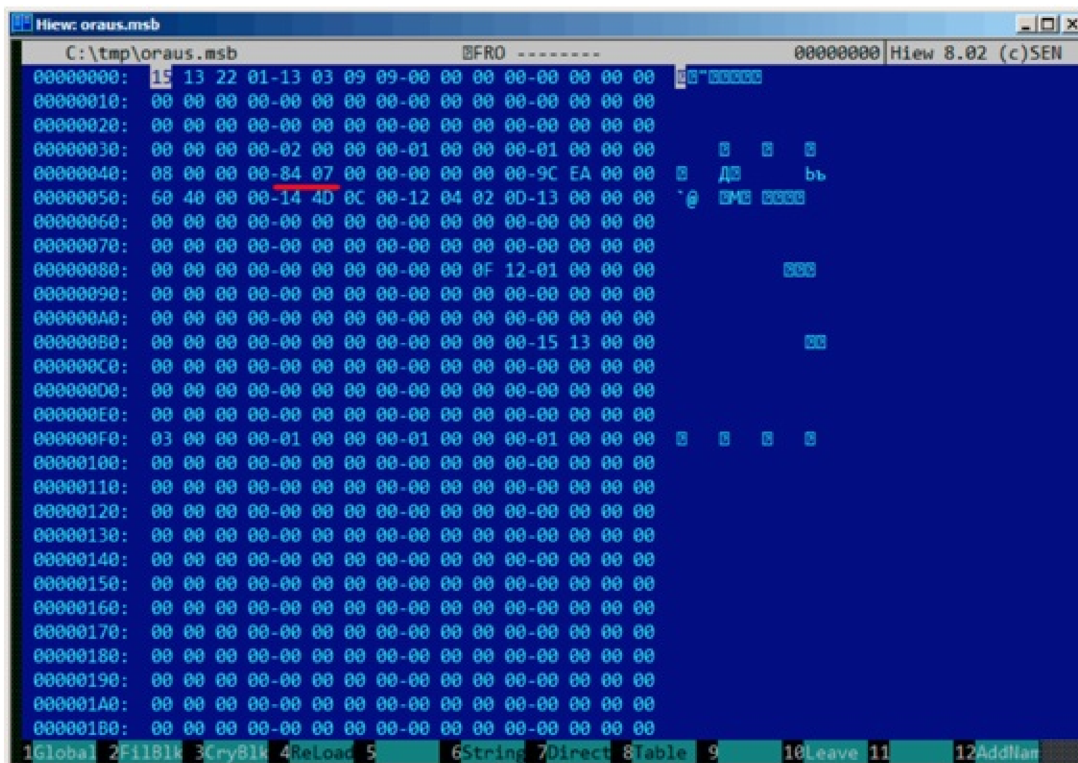
还有一点，Oracle RDBMS需要知道需要加载的字符串在块中的位置，对么？第一个字符串"normal, successful completion"从地址0x14444(如果我们从文件开始处计数的话)或者0x44(从块开始处计数)开始。第二个字符串"unique constraint (%s.%s) violated"从0x1461(从文件开始处计算)或者0x61(从块开始处计算)开始。这些数字(0x44和0x61)看上去很熟悉！我们能在块的开始处找到他们。

因此，每个6字节块是：

- 16比特错误码
- 16比特0(或者附加标志)
- 16比特当前块文本字符串起始位置

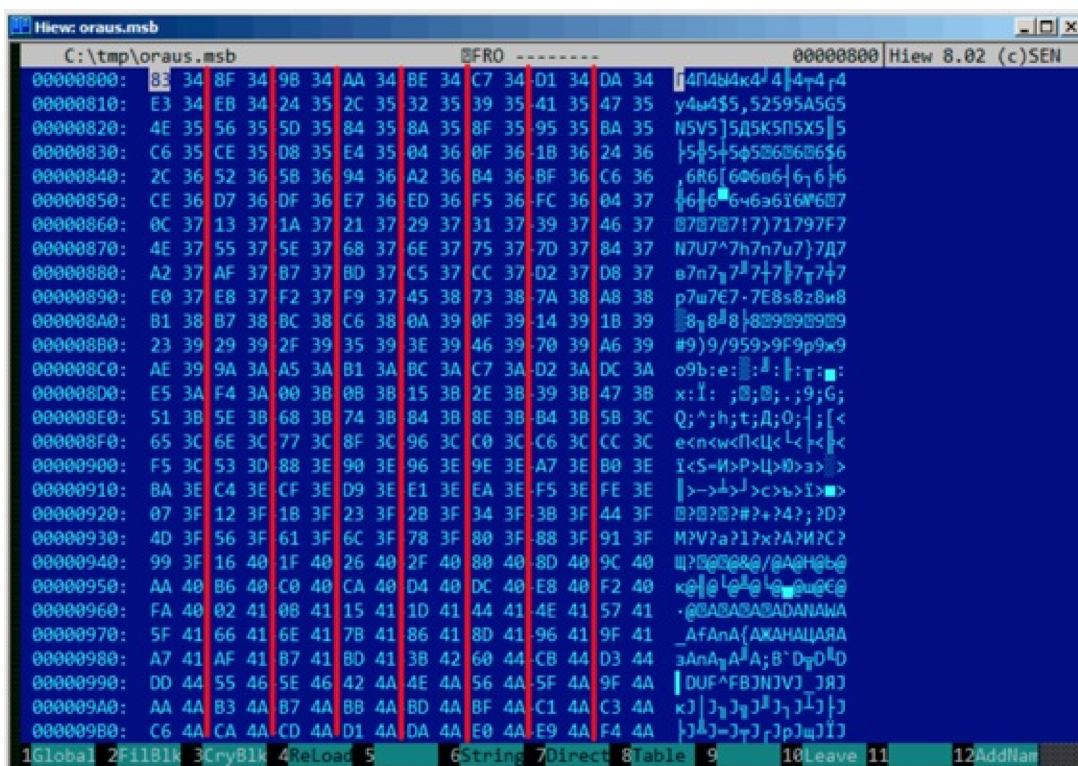
可以通过快速核对其他值证明我是对的。然后这里还有最后一个6字节块，错误码为0，从最后一条错误信息的最后一个字符后开始。也许这就是确定文本信息长度的方法?我们刚刚枚举了6字节块来寻找我们需要的错误码，然后我们找到了文本字符串的位置，接着我们通过查看下一个6字节块获取文本字符串的位置。这样我们就找到了字符串的边界。这种方法通过不保存文本字符串的大小节省了一些空间。我不敢说它特别省，但是这是一个聪明的技巧。

我们再回到.MSB文件的头部：



可以迅速找到文件中记录块数量的值(用红线标注出来了)，然后检查了其他.MSB文件，结果发现都是这样的。这里还有很多其他值，但我没有查看他们，因为我的工作已经完成了(一个unpack工具)。如果我要写一个.MSB文件packer，那么我可能需要理解其他值的含义。

头的后面接着一个可能包含16比特值的表：



其大小可以直观的划出来(我用红线画出)。在dump这些值的过程中，我发现每个16比特的值是每个块最后一个错误码。

这就是如何快速找到Oracle RDBMS错误信息的方法：

- 加载那个我称为last_errnos的表(包含每个块最后一个错误码)；
- 找到包含我们所需错误码的块，假定所有的错误码的增加跨越了每个块到所有文件；
- 加载特殊块；
- 枚举6字节结构体直到找到目标错误码；
- 从下一个6字节块获取最后一个字符的位置；
- 加载这个范围内错误信息所有的字符。

这是我编写的unpack.MSB文件的c程序：beginners.re

这是我用作实例的两个文件(Oracle RDBMS 11.1.0.6):beginners.re,beginners.re

87.1 总结

这种方法对于许多现代计算机来说也许太老了，假如这个文件格式是80年代中期某个具有内存/硬盘空间节省意识的硬件开发者设计的。尽管如此，这仍是一个有趣又简单的任务，因为不需要分析Oracle RDBMS的代码就能理解特殊文件的格式。

后记

附录